Chapter 1

Java Programming Fundamentals

Basic Parts of a Computer

- A Central Processing Unit (CPU)
- Memory
- Input/output devices

Bits and Bytes

- A *bit* is a binary digit (0 or 1).
- A *byte* is typically 8 bits.
- All programs and data are stored in the computer as sequences of bits.

Programming Languages

- High-level languages are easy for humans to read and write (e.g., Java).
- Low-level languages are closer to the hardware (e.g., machine or object code).
- *Compilers* are programs that translate high-level languages into object code.

Java

- Java is a high-level language conceived by James Gosling in 1991.
- It was named “Java” in 1995.
- Its syntax is similar to the older languages C and C++.
- It is an important language for the Internet.
  - Applets
  - Servlets
- The newest version is Java SE 7.

Java Compilers and the JVM

- Java compilers do not generate machine code for a CPU.
- Java compilers generate machine code for the JVM (Java Virtual Machine).
- The JVM machine code (called *bytecode*) is executed by a JVM interpreter program on each computer.

Object-Oriented Programming

- Code is organized around the data.
- You define the data and the routines that are permitted to act on the data.
- Java is object-oriented.
- In Java, a *class* encapsulates the data and the routines acting on the data.

First Simple Java Program

```
/*
   This is a simple Java program.

   Call this file Example.java.
*/
class Example {
  // A Java program begins with a call to main().
  public static void main(String[] args) {
    System.out.println("Java drives the Web.");
  }
}
```

Compiling and Running the Program

- Save the program to a file named `Example.java`
- To compile the program, type in

```
javac Example.java
```

  on the command line.
- To run the compiled bytecode, type in

```
java Example
```

A Second Example

```
class Example2 {
  public static void main(String[] args) {
    int var1; // this declares a variable
    int var2; // this declares another variable

    var1 = 1024; // this assigns 1024 to var1
    System.out.println("var1 contains " + var1);
    var2 = var1 / 2;
    System.out.print("var2 contains var1 / 2: ");
    System.out.println(var2);
  }
}
```

A Third Example

```
class Example3 {
  public static void main(String[] args) {
    int w; // declare an int variable
    double x; // declare a floating-point variable

    w = 10; // assign w the value 10
    x = 10.0; // assign x the value 10.0
    System.out.println("Original value of w: " + w);
    System.out.println("Original value of x: " + x);
    System.out.println(); // print a blank line
    // now, divide both by 4
    w = w / 4;
    x = x / 4;
    System.out.println("w after division: " + w);
    System.out.println("x after division: " + x);
  }
}
```

Converting Gallons to Liters

```
class GalToLit {
  public static void main(String[] args) {
    double gallons; // holds the number of gallons
    double liters; // holds conversion to liters

    gallons = 10; // start with 10 gallons

    liters = gallons * 3.7854; // convert to liters

    System.out.println(gallons + " gallons is " +
                       liters + " liters.");
  }
}
```

The if Statement

- Simplest form:
  if (*condition*) *statement*;
- Example:

```
if(3 < 4) System.out.println("yes");
```

- Relational operators:
  <, >, <=, >=, ==, !=

Example of if Statements

```
class IfDemo {
  public static void main(String[] args) {
    int a, b;

    a = 2;
    b = 3;

    if(a < b) System.out.println("a is less than b");

    // this won't display anything
    if(a == b)
      System.out.println("you won't see this");
  }
}
```

The for Loop

- General form:

  for(*initialization* ; *condition* ; *iteration*)

  statement;

- *Initialization* is normally for setting a loop control variable.
- *Condition* is for stopping the loop.
- *Iteration* is normally for updating the loop control variable.

Example of a for Loop

```
class ForDemo {
  public static void main(String[] args) {
    int count;

    for(count = 0; count < 5; count = count+1)
      System.out.println("This is count: " + count);

    System.out.println("Done!");
  }
}
```

Code Block

- A list of statements inside braces
- Can be used any place a single statement can be used
- Does not need to end in a semicolon
- Example:

```
if(w < h) {
   v = w*h;
   w = 0;
}
```

Indentation Practices

- The Java compiler doesn't care about indentation.
- Use indentation to make your code more readable.
- Indent one level for each opening brace and move back out after each closing brace.

Java Identifiers

- An identifier is a name given to a method, variable, or other user-defined item.
- Identifiers are one or more characters long.
- The dollar sign, the underscore, any letter of the alphabet, and any digit can be used in identifiers.
- The first character in an identifier cannot be a digit.
- Upper case and lower case are different: **myvar** and **MyVar** are different identifiers.

Chapter 2

Introducing Data Types and Operators

Java Types

- Java is a strongly typed language; that is, the compiler type-checks all statements.
- Two categories of types:
  - Primitive
  - Object/Reference

Java Primitive Types

- **boolean** (true/false)
- **char** (characters)
- **float** (single-precision floating point numbers)
- **double** (double-precision floating point)
- **byte** (8-bit integers)
- **short** (16-bit integers)
- **int** (32-bit integers)
- **long** (64-bit integers)

Ranges of Values for Integer Types

- byte: -128 to 127
- short: -32,768 to 32,767
- int: -2,147,483,648 to 2,147,483,647
- long: -9,223,372,036,854,775,808 to
  9,223,372,036,854,775,807

Example Using long Integers

```
class Inches {
  public static void main(String[] args) {
    long cubicInches, inchesPerMile;

    // compute the number of inches in a mile
    inchesPerMile = 5280 * 12;
    // compute the number of cubic inches
    cubicInches = inchesPerMile * inchesPerMile *
                  inchesPerMile;
    System.out.println("There are " + cubicInches +
                  " cubic inches in a cubic mile.");
  }
}
```

Output: There are 254358061056000 cubic inches in a cubic mile.

Floating Point Types

- float
  - 32 bits wide
  - approximately 7 decimal places of accuracy
  - range of values is approximately

    -3.4×10^{38} to $+3.4 \times 10^{38}$.
- double:
  - 64 bits wide
  - approximately 15 decimal places of accuracy
  - range of values is approximately

    -1.8×10^{308} to $+1.8 \times 10^{308}$

Examples Using Floating Point Type

```
double x;
x = 3.1416;
x = 0.;
x = 6.02E23; // 6.02 x 10^23
x = -3.6E-4;  // -3.6 x 10^-4
```

Characters

- Java uses *Unicode*.
- A **char** uses 16 bits with a range of values of 0 to 65,535.
- Each value represents a different character.
- The ASCII character set is a subset.
- Use single quotes to denote characters.
- Example:

```
char ch;
ch = 'A';
ch++; // now ch = 'B'
ch = 90; // now ch = 'Z'
```

Booleans

- Represents true/false values
- The words **true** and **false** are reserved words.
- Examples:

```
boolean b;
b = true;
if(b) System.out.println("yes");
b = (3 > 4);  // assigns false to b
```

Literals

- Literals refer to fixed values or constants.
- Examples: 'A', 23, 23.45, true, "true"
- The only boolean literals are **true** and **false.**
- Floating point literals include a decimal point and/or an exponent.
- You can have hexadecimal (base 16), octal (base 8) and binary (base 2) literals as well. (Binary literals are new in JDK 7.)

Character Literals

- Surround a character with single quotes.
- Special characters need escape sequences:

  single quote: \' tab: \t
  double quote: \" backspace: \b
  backslash: \\ newline: \n
  carriage return: \r

- Example:

```
char ch;
ch = '\t';
```

String Literals

- Surround a sequence of characters with double quotes.
- Examples:

```
String s;
s = "abc";
s = "abc\ndef";
s = "";  // empty string
```

Declaring Variables

- To declare a variable use the form:

  type variablename;

- You can declare several variables of the same type at once:

```
int x, y, z;
```

- You can initialize variables when you declare them:

```
int x = 3;
int y = 4, z = 5, w;
```

Scope of Variables in Methods

- A variable can be declared within any code block in a method.
- The *scope* of a variable is the part of the program in which the variable can be used.
- In general, a variable's scope consists of the code from where it is declared to the end of the code block in which it was declared.

Arithmetic Operators

- addition: +
- subtraction: -
- multiplication: *
- division: /
- modulus: %
- increment: ++
- decrement: --

Arithmetic Operator Examples

- 9 + 4 yields 13
- 9 – 4 yields 5
- 9 * 4 yields 36
- 9 / 4 yields 2 (when two integers are divided, the remainder is thrown away)
- 9.0/4 yields 2.25
- 9 % 4 yields 1 (the remainder when 9 is divided by 4)
- x++ and ++x increment the value of x by 1
- x-- and --x decrement the value of x by 1

Relational Operators

Operator	Meaning
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Logical Operators

Operator	Meaning
&	AND
|	OR
^	XOR (exclusive OR)
||	short-circuit OR
&&	short-circuit AND
!	NOT

Truth Table

p	q	p & q	p | q	p ^ q	! p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

Assignment Operator

- General form:

  var = expression;

- Variations:

```
x = y = z = 3;
x += 10; // adds 10 to x
```

- More compound assignment operators:

+=	-=	*=	/=
%=	&=	!=	^=

Automatic Type Conversions

- A variable of one type A can be assigned a value of another type B if:
  - the two types are compatible
  - the type A is larger than the type B
- In that case, the value is automatically converted from type B to type A.
- Example:

```
double x = 3;  // the integer 3 is converted to
               // the double 3.0
```

Type Conversion

- A variable of type A can be assigned a value of type B even if it isn't a widening conversion if it is compatible and you use a type cast.
- Example:

```
int x = (int) 3.14;  // assigns 3 to x
byte y = (byte) 256; // assigns 0 to b
```

Operator Precedence

High

++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
? :						
=	*op*=					

Low

Chapter 3

Program Control Statements

Inputting chars from the Keyboard

- **System.in.read( )** will wait until the user presses a key and then presses ENTER.
- It then returns the character as an **int**.
- Example:

```
char ch;
ch = (char) System.in.read();
System.out.println(ch);
```

The if Statement

- Complete form:
  if (*condition*) *statement*;
  else *statement*;
- The two target statements can be code blocks.
- You can nest **if** statements to form a ladder:

```
if(x > y) System.out.println("x");
else if(y > z) System.out.println("y");
else if(x > z) System.out.println("z");
else if(x == z) System.out.println("a");
else System.out.println("z");
```

Switch Statement General Form

```
switch( expression ) {
    case constant1:
      statement sequence
      break;
    case constant2:
      statement sequence
      break;
    case constant3:
      statement sequence
      break;
    ...
    default:
      statement sequence
}
```

The for Statement

- General form:

  for (*intialization* ; *condition* ; *iteration*)

  statement;

- Examples:

```
for(int x = 10; x > 0; x--)
  System.out.println(x);

for(int x = 0, int y = 0; x+y < 10; x++, y++)
  total += x + y;

for( ; ; )
  System.out.println("Infinite loop");
```

The while Statement

- General form:
  while (*condition*) *statement;*
- Examples:

```
while(x < 10) {
  System.out.println(x);
  x++;
}

while(true)
  System.out.println("Infinite loop");
```

The do-while Statement

- General form:

  do {

  statements;

  } while (*condition*);

- Example:

```
char ch;
do {
  ch = (char) System.in.read();
  System.out.println(ch);
} while(ch != 'x');
```

The break Statement

- The **break** statement can be used to exit loops.
- Example:

```
while(x > 0) {
  System.out.println(x);
  x++;
  if(x > 100) break;
}
```

The continue Statement

- The **continue** statement inside a loop causes the rest of the body of the loop to be skipped and a jump to occur directly to the conditional statement to begin the next iteration of the loop.
- Example:

```
int x = 0;
while(x < 10) {
  x++;
  if(x % 2 == 0) continue;
  System.out.println(x);
}
```

Chapter 4

Introducing Classes, Objects, and Methods

Class Fundamentals

- Classes are templates or blueprints that specify how to build objects.
- Objects are instances of a class.
- A class can be used to create any number of objects, all of the same form, but possibly containing different data.
- Well-designed classes group logically connected data with methods for acting on that data.

Class General Form

```
class classname {
    // declare instance variables
    type varname;

    // declare constructors
    classname( parameters ) {
      // body of constructor
    }

    // declare methods
    type methodname( parameters ) {
      // body of method
    }
}
```

Example of a Class

- **Vehicle** class declaration:

```
class Vehicle {
  int passengers, fuelCap, mpg;
}
```

- **VehicleDemo** class declaration:

```
class VehicleDemo{
  public static void main(String[] args) {
    Vehicle van = new Vehicle();
    Vehicle car = new Vehicle();
    car.mpg = 25;
    car.fuelCap = 12;
  }
}
```

Reference Variables

- The local variables **van** and **car** refer to different objects.
- Both objects have the same form (with 3 instance variables **passengers**, **fuelCap**, and **mpg**), but they have their own copies of the 3 instance variables.
- To access instance variables and methods, use the dot (.) operator:

```
car.mpg = 25;
car.fuelCap = 12;
```

Assignment of References

- If you assign

```
van = car;
```

  then both variables refer to the same object.
- In that case, if you change an instance variable's value in **`van`**, it will change the value in **`car`** as well, since they refer to the same object.
- The object that **`van`** previously referred to is garbage collected if no other references to the object exist.

Methods

- General form:

```
return-type methodname ( parameters ) {
    statements;
}
```

- Parameters are local variables that receive their values from the caller of the method.
- The return type can be any valid type or **void** if the method doesn't return a value.

Example Method

- The following method can be added to the **Vehicle** class:

```
void range() {
  System.out.println("range: " + fuelCap * mpg);
}
```

- If the **VehicleDemo** class's **main( )** method calls

```
car.range();
```

then "range: 300" will be displayed.

Returning from a **void** Method

- Two ways to return from a **void** method:
  - when the method's closing brace is encountered
  - when a **return** statement is encountered
- Examples:

```
void sayHello() {
  System.out.println("Hello");
}

void sayHello() {
  System.out.println("Hello");
  return;
}
```

Returning a Value

- To return a value from a method, you must use a **return** statement of the form:

  return *value*;

- Example:
  - In the **Vehicle** class:

```
int range() {
   return fuelCap * mpg;
}
```

  - In **VehicleDemo** class's **main( )** method:

```
int range = car.range();
System.out.println(range); // prints "300"
```

Using Parameters

- A *parameter* is a variable whose scope is the method body and whose initial value is specified by the caller.
- Example:
  - In the **Vehicle** class:

```
double fuelNeeded(int distance) {
  return (double) distance / mpg;
}
```

  - In **VehicleDemo**'s **main( )** method:

```
System.out.println(car.fuelNeeded(750));
// prints "30.0"
```

Constructors

- Used to initialize an object when it is created
- Syntactically it is like a method.
- Its name is the name of the class.
- Example constructor for **Vehicle** class:

```
Vehicle() {
  fuelCap = 12;
  mpg = 25;
  passengers = 5;
}
```

Constructors with Parameters

- Example:
  - In the **Vehicle** class:

```
Vehicle(int p, int f, int m) {
  passengers = p; fuelCap = f; mpg = m;
}
```

  - In the **VehicleDemo**'s **main( )** method:

```
Vehicle car = new Vehicle(5, 12, 25);
Vehicle van = new Vehicle(7, 24, 21);
```

The Keyword **this**

- **this** is an implicit argument that refers to the object on which the method is called.
- **this** is useful for making it clear that you are referring to an instance variable.
- Example in **Vehicle** class:

```
double fuelNeeded(int distance) {
  return (double) distance / this.mpg;
}
```

Chapter 5

More Data Types and Operators

Arrays

- An *array* is a collection of variables of the same type referred to by a common name.
- Each of the variables is specified by an index.
- One way to declare an array:

  type[ ] *arrayname* = new *type*[*size*];
- Example:

```
int[] grades = new int[30];
```

- In the example, **grades** is the name of the collection of 30 integer variables.

Accessing Array Variables

- You must specify an index to access a variable.
- If the array has size 30, the indices are 0 to 29.
- Example:

```
int[] grades = new int[30];
grades[0] = 100;
grades[29] = 0;
grades[1] = grades[0];
grades[2] = grades[3*2+1];
for(int i = 0; i < 30; i++)
  grades[i] = i+70;
```

Array Initializers

- To create and initialize an array at the same time, you can use this form:

  type[ ] *arrayname* = { *val1*, *val2*, …, *valN* };
- Example:

```
int[] fourVals = {3, 1, 4, 1};
```
- This example creates an array of length 4 storing the four values 3, 1, 4, and 1.

Bubble Sort

- You can sort an array **nums** of length **size** as follows:

```
for(int a = 1; a < size; a++)
  for(int b = size-1; b >= a; b--) {
    if(nums[b-1] > nums[b]) { // if out of order
      // exchange elements
      int t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
  }
}
```

Two-dimensional Arrays

- A two-dimensional array is an array of one-dimensional arrays.
- Examples:

```
int[][] table = new int[3][4];
table[0][1] = 3;
for(int i = 0; i < 3; i++)
  for(int j = 0; j < 4; j++)
    table[i][j] = i+j;

int[][] newTable = {{1,2},{3,4},{5,6}};
```

Irregular Two-dimensional Arrays

```
int[][] data = new int[3][];
data[0] = new int[1];
data[1] = new int[2];
data[2] = new int[4];

int[][] moreData = {{1},{2,3},{4,5,6}};
```

Array **length** Member

- All arrays have a read-only instance variable called **length**.
- Example:

```
int[] data = new data[5];
for(int i = 0; i < data.length; i++)
  data[i] = 3*i;
```

The **for-each** Style Loop

- General form:
  for(*type iterVar* : *collection*) *statement-block*
- Example:

```
int[] data = {3,4,5,6};

// these two loops do the same thing
for(int i = 0; i < data.length; i++)
  System.out.println(data[i]);

for(int v : data)
  System.out.println(v);
```

Constructing Strings

- Strings are objects of class **String**.
- Example:

```
// all 3 statements create new String objects
String s1 = "hello";
String s2 = new String("hello");
String s3 = new String(s2);
```

Some Operations on **Strings**

boolean equals(*str)*	Returns true if the invoking string contains the same character sequence as *str*.
int length( )	Returns the number of characters in the string.
char charAt(*index*)	Returns the character at the index specified by *index*.
int compareTo(*str*)	Returns a negative value if the invoking string is less than *str*, a positive value if the invoking string is greater than *str*, and zero if the strings are equal.
int indexOf(*str*)	Searches the invoking string for the substring specified by *str*. Returns the index of the first match or -1 on failure.

Examples Using **String** Operations

```
String s1 = "abcde";
System.out.println(s1.length()); // prints "5"
System.out.println(s1.charAt(2)); // prints "c"
if(s1.compareTo("xyz") < 0)
  System.out.println("Yes"); // prints "Yes"
System.out.println(s1.indexOf("bc")); // prints "1"
System.out.println(s1.indexOf("f")); // prints "-1"
System.out.println(s1.indexOf("ef")); // prints "-1"
```

String Properties

- **String**s are immutable; you can read their characters, but you can't change them.
- In JDK 7, you can use a **String** to control a **switch** statement:

```
switch(s) {
  case "b":
    System.out.println(s);
    break;
  case "c":
    System.out.println(s);
    break;
  ...
```

Bitwise Operators

Operator	Result
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Unsigned shift right
<<	Shift left
~	Bitwise NOT

Bit Actions

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

The ? Operator

- General form:
  condition ? *expression1* : *expression2*
- This entire form is an expression whose value is the value of *expression1* if *condition* is true and whose value is the value of *expression2* if *condition* is false.
- Examples:

```
int x = (3 < 4 ? 5 : 6); // assigns 5 to x
int abs = (x < 0 ? -x : x); // abs is assigned the
                            // absolute value of x
```

Chapter 6

A Closer Look at Methods and Classes

public and private Access Modifiers

- Public members can be accessed by any other code in your program.
- Private members can be accessed only by other members of its class.
- Example:

```
class A { public int x; private int y; }
class B {
  public static void main(String[] args) {
    A a = new A();
    a.x = 3; // legal
    a.y = 3; // ILLEGAL - compiler error
  }
}
```

Passing Arguments to Methods

- In Java, arguments are passed using *call-by-value*.
- The value of the argument is copied into the parameter of the method.
- If the argument is primitive, then the primitive value is copied and so changes made to the parameter do not affect the argument.
- If the argument is a reference type, then the reference is copied and so the parameter and the argument refer to the same object.

Example of Passing Primitives

```
class Data {
  void make3(int x) {
    x = 3;
  }
}
class Test {
  public static void main(String[] args) {
    Data d = new Data();
    int y = 4;
    d.make3(y);
    System.out.println(y);  // prints 4
  }
}
```

Example of Passing References

```
class Data {
  public int x;
  void addTo(Data d) { // adds this x to d.x
    d.x = d.x + this.x;
  }
}
class Test {
  public static void main(String[] args) {
    Data d1 = new Data(), d2 = new Data();
    d1.x = 3; d2.x = 4;
    d1.addTo(d2);
    System.out.println(d2.x); // prints 7
  }
}
```

Method Overloading

- Two methods in a class can share a name, as long as they have different parameter declarations.
- Example:

```
class Overload {
  void display() {
    System.out.println("<nothing>");
  }
  void display(int x) {
    System.out.println(x);
  }
}
```

Constructor Overloading

- You can overload constructors.
- Example:

```
class Overload{
  int data;
  Overload(int x) {
    data = x;
  }
  Overload(int x, int y) {
    data = x+y;
  }
}
```

Recursion

- A method can call itself—this is called *recursion*.
- Each call of the method has its own copies of the method's variables that are kept on a stack.
- To avoid having a method call itself forever, there must be a *base case* that is eventually reached where the method does not call itself.

Recursion Example 1

```
void drawStars(int n) {
  if(n == 1) // base case
    System.out.print("*");
  else {     // recursive case
    System.out.print("*");
    drawStars(n-1); // a recursive call
  }
}
```

Recursion Example 2

```
// print the contents of an array
void printArray(int[] array) {
  printArrayAux(array, 0); // start at index 0
  System.out.println();
}

void printArrayAux(int[] array, int index) {
  if(index == array.length)
    return; // we are done
  else { // there are more elements to print
    System.out.print(array[index] + " ");
    printArrayAux(array, index+1);
  }
}
```

Recursion Example 3

- The *factorial* of an integer *n* is (*n*)*(*n*-1)*(*n*-2)*...*3*2*1. For example, the factorial of 5 is 5*4*3*2*1 = 120.
- Recursive factorial method:

```
int fact(int n) {
    int result;
    if(n == 1) // base case
      return 1;
    else {     // recursive case
      result = fact(n-1) * n;
      return result;
    }
 }
```

Static Variables

- If you declare a variable **static**, it is a variable shared by all instances of the class.
- Example:

```
class StatC {
  int x = 3;  // instance variable, not static
  static int y = 4; // static variable

  public static void main(String[] args) {
    StatC s1 = new StatC(), s2 = new StatC();
    s1.x = 5; // changes s1's x, not s2's x
    s1.y = 5; // changes y for both s1 and s2
  }
}
```

Static Methods

- If you declare a method **static**, it can be called independently of any object through its class name.
- Example:

```
class StatMethod {
  static void statMeth() {
    System.out.println("Called");
  }
}
class Test {
  public static void main(String[] args) {
    StatMethod.statMeth(); // prints "Called"
  }
}
```

Inner Classes

- A class declared within another class is called an *inner class*.
- Its scope is the enclosing class.
- It has access to all the variables and methods of the enclosing class.

Example of an Inner Class

```
class Outer {
  int x = 5;

  class Inner {
    void changeX() { x = 3; } // change Outer's x
  }

  void adjust() {
    Inner inn = new Inner();
    inn.changeX(); // Outer's x is now 3
  }
}
```

Varargs

- You can create methods with a variable number of arguments, using "..."
- Example:

```
class VATest {
  static void vaMethod(int ... data) {
    for(int x : data) System.out.print(x + " ");
  }
  public static void main(String[] args) {
    vaMethod(1); // prints "1 "
    vaMethod(1,2,3); // prints "1 2 3 "
  }
}
```

Overloading Varargs Methods

- You can overload varargs methods.
- Ambiguity can result in some cases.
- Example:

```
class VAOverload {
  static void vaMethod(int ... data) { }
  static void vaMethod(boolean ... data) { }
}
```

- This is fine unless you call **vaMethod( )** with no arguments, which is ambiguous.

Chapter 7

Inheritance

Inheritance

- A class (a "subclass") can inherit all the methods and variables of another class (a "superclass").
- Use the keyword **extends**.
- General form:

  class *subclass* extends *superclass* { ... }
- The subclass extends the superclass by adding behavior and data to the behavior and data provided by the superclass.

Inheritance Example

```
class OneDimPoint {
  int x = 3;
  int getX() { return x; }
}
class TwoDimPoint extends OneDimPoint {
  int y = 4;
  int getY() { return y; }
}
class TestInherit {
  public static void main(String[] args) {
    TwoDimPoint pt = new twoDimPoint();
    System.out.println(pt.getX() + "," + pt.getY());
  }
}
```

Properties of Inheritance

- A subclass cannot access the private members of its superclass.
- Each class can have at most one superclass, but each superclass can have many subclasses.
- A subclass constructor can call a superclass constructor by use of **super( )**, before doing anything else.
- If you do not call a superclass constructor, the no-argument constructor is automatically called.

Example 1 of Subclass Constructors

```
class OneDimPoint {
  int x;
  OneDimPoint() { x = 3; }
  int getX() { return x; }
}
class TwoDimPoint extends OneDimPoint {
  int y;
  TwoDimPoint() { y = 4; } // automatically calls
                           // OneDimPoint() first
  int getY() { return y; }
}
```

Example 2 of Subclass Constructors

```
class OneDimPoint {
  int x;
  OneDimPoint(int startX) { x = startX; }
  int getX() { return x; }
}
class TwoDimPoint extends OneDimPoint {
  int y;
  TwoDimPoint(int startX, int startY) {
    super(startX); // explicitly calls constructor
    y = startY;
  }
  int getY() { return y; }
}
```

Using super to Access Members

- You can use **super** similar to **this** to access superclass members from the subclass.
- Example:

```
class OneDimPoint {
  int x = 3;
}
class TwoDimPoint extends OneDimPoint {
  int x = 4;
  int getSum() { return this.x + super.x; }
}
```

Reference Variables and Subclasses

- To assign a value to a variable, the type of the value must match the type of the variable.
- Example:

```
int x = 3; // OK
int x = "Hello"; // ERROR--type mismatch
```

- However, you can assign an object of a subclass to a variable of a superclass type.
- Example:

```
OneDimPoint pt = new TwoDimPoint();
```

Method Overriding

- A subclass can declare a method with the same return type and signature as an inherited method.
- In that case, we say the subclass method *overrides* the inherited superclass method.
- When an overridden method is called from within a subclass, it will always refer to the version defined by that subclass—the superclass version is hidden.

Example of Overriding

```
class OneDimPoint {
  int x = 3;
  int getX() { return x; }
}
class TwoDimPoint extends OneDimPoint {
  int y = 4;
  int getX() { return y; } // overriding!
}
class TestOverriding {
  public static void main(String[] args) {
    OneDimPoint pt = new TwoDimPoint();
    System.out.println(pt.getX()); // prints "4"
  }
}
```

Dynamic Method Dispatch

- If a variable of a superclass type refers to a subclass object and an overridden method is called, the subclass's version is executed.
- In summary, the version of an overridden method that is executed depends on the class of the object being referenced.
- So the version to be executed is chosen at runtime.

Example of Dynamic Dispatch

```
class OneDimPoint {
  int getX() { return 3; }
}
class TwoDimPoint extends OneDimPoint {
  int getX() { return 4; } // overriding!
}
class TestDispatch {
  public static void main(String[] args) {
    OneDimPoint[] pts =
      {new TwoDimPoint(), new OneDimPoint()};
    for(OneDimPoint pt : pts)
      System.out.print(pt.getX()); // prints "43"
  }
}
```

Abstract Classes

- Sometimes you want a class that is only partially implemented and you want to leave it to the subclasses to complete the implementation.
- In that case, use an *abstract* class with *abstract* methods.
- To declare a class or method as abstract, just add the keyword **abstract** in front of the class or method declaration.
- In the case of an abstract method, you must also leave off the body of the method.

Example of an Abstract Class

```
abstract class Super {
  int x;

  int getX() { return x; }

  abstract void setX(int newX); // no body
}

class Sub extends Super {

  void setX(int newX) { x = newX; }
}
```

The Keyword final

- If you do not want a class to be subclassed, precede the class declaration with the keyword **final**.
- If you do not want a method to be overridden by a subclass, precede the method declaration with the keyword **final**.
- If you want a variable to be read-only (that is, a constant), precede it with the keyword **final**.

Example Using final

```
final class MyClass {
  final int x = 3;
  public static final double PI = 3.14159;

  final double getPI() { return PI; }
}
```

The Object Class

- Java defines a special class called **Object** that is an implicit superclass of all other classes.
- Therefore, all classes inherit the methods in the **Object** class.
- A variable of type **Object** can refer to an object of any other class, including an array.

Some Methods in the Object Class

Method	Purpose
Object clone( )	Creates a copy of this object
boolean equals(Object *obj*)	Tests whether two objects are equal
void finalize( )	Called before recycling the object
Class<?> getClass( )	Returns the class of the object
int hashCode( )	Returns the hash code of the object
void notify( )	Resumes execution of a thread waiting on the object
void notifyAll( )	Resumes execution of all threads waiting on the object
String toString( )	Returns a string describing the object
void wait( )	Waits on another thread of execution

Chapter 8

Interfaces

Interface Fundamentals

- An **interface** defines a set of methods that will be implemented by a class.
- Syntactically, interfaces are similar to abstract classes, except that no method can include a body.
- Any number of classes can implement the interface, and any class can implement any number of interfaces.
- All the methods in an interface are implicitly public.
- Use the keyword **implements** to indicate your class implements the interface.

Example of an Interface

```
public interface Series {
  int getNext();
  void reset();
  void setStart(int x);
}

public class WholeNumberSeries implements Series {
  int value;
  public int getNext() { value++; return value;}
  public int getPrior() { return value; }
  public void reset() { value = 0; }
  public void setStart(int x) { value = x; }
}
```

Using Interface References

- Each interface defines a new type, and so variables can be declared to be of an interface type.
- Such variables can refer to an object of any class that implements the interface.
- Example:

```
Series s = new WholeNumberSeries();
s.reset();
```

Constants in Interfaces

- Interfaces can also include data members, but they are implicitly **public final static** variables and must be initialized.
- Thus they are essentially constants.
- These constants are accessed like other static variables.
- Example:

```
interface MathConstants {
  double PI = 3.14159;
  double E = 2.71828;
}
```

Extending Interfaces

- An interface can extend another interface using the keyword **extends**.
- A class implementing the subinterface must implement all the methods in that interface and the superinterface.

Nested Interfaces

- An interface can be declared a member of another interface or of a class.
- In that case, it is called a *member* or *nested* interface.
- Such an interface can be declared **public**, **private**, or **protected**.

Chapter 9

Packages

Package Fundamentals

- A *package* is a group of related classes and interfaces.
- Packages provide a way of organizing your code and of controlling access to the code.
- Classes defined within a package must be accessed through their package name.
- When no package is specified, the global (default) no-name package is used.

Defining a Package

- General form:
  package *pkgname*;
- This statement must be at the top of a Java source file.
- Usually lower case is used for the package name.
- In the file system, each package is stored in its own directory that has the same name as the package.

Using Packages

- You can define packages inside of packages.
- To put a class in a package **pkg2** which is nested inside a **pkg1**, use the statement:

  package pkg1.pkg2;
- To compile **MyClass.java** that is in the package **pkg1.pkg2**, use

  javac pgk1/pgk2/MyClass.java
- To run **MyClass**, use

  java pgk1.pgk2.MyClass

Importing Packages

- To use **MyClass**, you can use the fully qualified name, as follows:

```
pkg1.pkg2.MyClass v = new pkg1.pkg2.MyClass();
```

- Or use the **import** statement with general form:

  import *pkg.classname*;

- Use an asterisk (*) for the classname to import the entire contents of a package.
- The **import** statement goes after the **package** statement and before any class definitions.
- With imports, you need not qualify **MyClass**.

Packages and Member Access

	Private	Default	Protected	Public
Visible within same class	Yes	Yes	Yes	Yes
Visible within the same package by subclass	No	Yes	Yes	Yes
Visible within the same package by non-subclass	No	Yes	Yes	Yes
Visible within different packages by subclass	No	No	Yes	Yes
Visible within different packages by non-subclass	No	No	No	Yes

Commonly Used Standard Packages

Package	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit
java.util	Contains various utility classes, plus the Collections Framework

The **java.lang** package is imported automatically in every Java program. All others need to be explicitly imported.

Static Import

- The use of **import** followed by **static** allows you to import static members of a class.
- Those static members can then be referred to directly by their names, without having to qualify them with the name of their class.
- For example, to use the **Math** function **sqrt( )** in the form "sqrt( )" instead of "Math.sqrt( )", add one of the **import** statements

```
import static Math.sqrt;
import static Math.*;
```

Chapter 10

Exception Handling

Exceptions

- An *exception* is an error that occurs at runtime.
- Java defines standard exceptions for common program errors, such as dividing by zero.
- You can handle these exceptions in Java.
- All exceptions are represented by subclasses of **Throwable**.
- **Throwable** has subclasses **Error** and **Exception**.
- We will deal only with **Exception**s.

Exception Fundamentals

- Exception handling is managed via five keywords: **try, catch, throw, throws, finally**.
- Statements that might cause exceptions should be put in a **try** block.
- If an exception occurs, we say it is "thrown".
- You can catch the exception using **catch** and then handle it in some rational manner.

Using try and catch

- General form:

```
try {
   // block of code to monitor for errors
}
catch( ExceptionType exObj ) {
  // handle ExceptionType errors
}
... // catch blocks for other types of errors
```

More about try and catch

- If the code in the **try** block throws an exception, the **catch** block for the type of exception thrown is executed.
- If there is no **catch** block for the type of exception thrown, the exception is handled elsewhere (discussed later).
- If the code in the **try** block does not throw any exceptions, no **catch** block is executed.

Try-catch Example

```
try {
  int x = 3/y;
  System.out.println("Prints only if y is non-zero.");
}
catch(DivideByZeroException exc) {
  System.out.println("Tried to divide by zero.");
}
System.out.println("After catch.");
```

- If **y** is zero, then "Tried to divide by zero." is printed, followed by "After catch."
- If **y** is not zero, then "Prints only if y is non-zero." is printed, followed by "After catch."

Uncaught Exceptions

- All exceptions must be caught somewhere.
- If your method does not catch the exception where it occurs, the exception is thrown to the method (if any) that called your method.
- If that method does not catch it, it is thrown further up the call chain.
- If no code catches it, the JVM does, in which case it terminates execution and displays an error message.

Catching Multiple Exceptions

- With each **try** block, you can have several **catch** blocks for catching different kinds of exceptions.
- In the code below, the first **catch** block catches **DivideByZeroException**s and the second block catches all other subclasses of **Exception**.

```
try {
    // code that might throw an exception
} catch(DivideByZeroException ex1) {
  System.out.println("Divide by zero");
} catch(Exception ex2) {
  System.out.println("Some other exception");
}
```

Manually Throwing an Exception

- General form:

  throw *exceptionObject*;

- Example:

```
try {
  System.out.println("Before throw.");
  throw new ArithmeticException();
}
catch (ArithmeticException exc) {
  // catch the exception
  System.out.println("Exception caught.");
}
System.out.println("After try/catch block.");
```

Nested Blocks and Rethrowing

- A **try** block can be nested within another.
- An exception not caught by the inner **catch** block propagates to the outer **catch** block.
- Any **catch** block can rethrow the exception it catches or can throw a new exception.
- An exception thrown in a **catch** block will not be caught by that **catch** block.

Using the Thrown Exception Objects

- The **catch** block receives the exception object as a parameter.
- That exception object has methods inherited from **Throwable** that you can call to get further information about the exception.

Some Exception Methods

Method	Description
String getMessage( )	Returns a description of the exception
void printStackTrace( )	Displays the stack trace
String toString( )	Returns a **String** object containing a complete description of the exception.
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.

Using finally

- A **finally** block is always executed after a **try**/**catch** block is exited.
- It is useful for cleaning up afterwards, such as releasing resources.
- To specify a **finally** block, add it at the end of a **try**/**catch** sequence.

Example of a finally Block

```
try {
  int x = 3/0;
}
catch(DivideByZeroException exc) {
  System.out.println("Divide by zero");
}
finally {
  System.out.println("Leaving try");
}
```

Using throws

- **Error** and **RuntimeException** and their subclasses are called *unchecked* exceptions.
- If a method might throw and not catch a *checked* exception, then it must declare that it throws that kind of exception by use of **throws**.
- General form:

```
returnType methodName( params )
      throws exceptionList {
    // body
}
```

Java's Built-in Exceptions

- The **java.lang** package includes a number of subclasses of **RuntimeException**.
- They do not need to be declared in a **throws** clause of a method.
- The **java.lang** package also includes several **Exception** subclasses that need to be declared in a **throws** clause if they are not caught in the method.
- Other Java packages include many more exceptions for you to use.

Creating Exception Subclasses

- You can define your own subclass of **Exception** or any of its subclasses.
- Your subclass could override **toString( )** so that it gives more helpful information for your application.
- You throw your exceptions the same way you throw other exceptions.

Chapter 11

Using I/O

Java's I/O System

- The I/O classes discussed here support text-based console and file I/O.
- They are built on the concept of streams.
- A *stream* either produces or consumes information.
- All streams behave the same, regardless of the actual physical devices they are linked to.

Byte and Character Streams

- There are two hierarchies of stream classes in Java: byte and character.
- Byte streams are useful for file I/O and raw binary data.
- There are several predefined byte streams with **InputStream** and **OutputStream** as base classes.
- Character streams are useful for text-based I/O.
- There are several predefined character streams, with **Reader** and **Writer** as base classes.

Some InputStream Methods

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will throw an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. If the end of the stream is encountered, -1 is returned.
int read(byte[] *buffer*)	Attempts to read up to *buffer*.length bytes into buffer and returns the number of bytes that were read.
int read(byte[] *buffer*, int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes read.
long skip(long *numbytes*)	Ignores *numBytes* bytes of input, returning the actual number of bytes ignored.

Some OutputStream Methods

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
void write(int *b*)	Writes a single byte to an output stream.
void write(byte[ ] *buff*)	Writes a complete array of bytes to an output stream.
void write(byte[ ] *buff*, int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buff,* beginning at *buff*[*offset*].

Reading Console Input

- **System.in** is an **InputStream** you have automatic access to.
- **System.in.read( )** waits until the user presses a key and then returns the result as an **int**.
- You have to press ENTER after typing the key for it to be read.
- An **IOException** may be thrown.
- Example:

```
char ch = (char) System.in.read( );
```

Writing Console Output

- **System.out** is a **PrintStream** you have automatic access to.
- **System.out.print( )** and **System.out.println( )** display output to the console, as you have already seen.
- The **PrintStream** methods **write( ), printf( ),** and **format( )** can also be used.

Reading Files with Byte Streams

- Create a **FileInputStream** for the file and use its methods to read the file.

```
// reads and displays characters from a file
void displayFile(String fname) throws IOException {
  FileInputStream fin = new FileInputStream(fname);
  int i = fin.read();
  while(i != -1) {
    System.out.print((char) i);
    i = fin.read();
  }
  fin.close();
}
```

Writing to Files with Byte Streams

- Create a **FileOutputStream** for the file and use its methods to write to the file.

```
// write contents to a file with the given name
void writeFile(String fname, String contents)
    throws IOException {
  FileOutputStream fout = new FileOutputStream(fname);
  for(int i = 0; i < contents.length(); i++)
    fout.write(contents.charAt(i));
  fout.close();
}
```

Closing Files

- It is appropriate to move the call to **close( )** into a **finally** block to help ensure that it is executed, even if an exception occurs.
- JDK 7 has a **try**-with-resources statement that automates the closing of the resource after use.

Try-with-resources Example

```
void copyFile(String fromFile, String toFile) {
  try (FileInputStream fin =
                    new FileInputStream(fromFile);
       FileOutputStream fout =
                    new FileOutputStream(toFile)) {
    do {
        i = fin.read();
        if(i != -1) fout.write(i);
    } while(i != -1);

  } catch(IOException exc) {
    System.out.println("I/O Error: " + exc);
  }
}
```

Reading Binary Data

- The **DataInputStream** class allows you to read binary data of Java standard types.

```
void readTwoInts(String filename)
    throws IOException {
  DataInputStream din = new DataInputStream(
                new FileInputStream(filename));
  int x = din.readInt();
  int y = din.readInt();
  System.out.println(x + "," + y);
  din.close();
}
```

Writing Binary Data

- The **DataOutputStream** class allows you to write binary data of Java standard types.

```
void writeTwoInts(String filename)
    throws IOException {
  DataOutputStream dout = new DataOutputStream(
                new FileOutputStream(filename));
  dout.writeInt(3);
  dout.writeInt(4);
  dout.close();
}
```

Some Reader Methods

Method	Description
abstract void close( )	Closes the input source. Further read attempts will throw an **IOException**.
int read( )	Returns an integer representation of the next available character of input. If the end of the stream is encountered, -1 is returned.
int read(char[] *buffer*)	Attempts to read up to *buffer*.length characters into *buffer* and returns the number of **char**s that were read.
int read(char[] *buffer*, int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters read.
int read(CharBuffer *buffer*)	Attempts to fill the buffer specified by *buffer*, returning the number of characters successfully read.
long skip(long *numChars*)	Ignores *numChars* characters of input, returning the actual number of characters ignored.

Some Writer Methods

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the output stream. Returns a reference to the stream.
Writer append (CharSequence *chars*)	Appends *chars* to the end of the output stream. Returns a reference to the stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
void write(int *ch*)	Writes a single character to the output stream.
void write(char[ ] *buffer*)	Writes a complete array of characters to the output stream.
void write(String *str*)	Writes *str* to the output stream.

Console Input Using Char Streams

```
void echoInputUntilStop()
    throws IOException {
  BufferedReader br =
            new BufferedReader(
                 new InputStreamReader(System.in));
  String line = br.readLine();
  while(! line.equals("Stop")) {
    System.out.print(line);
    line = br.readLine();
  }
  br.close();
}
```

Console Output Using Character Streams

```
void printValues() {
  PrintWriter pw = new PrintWriter(System.out, true);
  int i = 10;
  double d = 123.65;

  pw.println("Using a PrintWriter.");
  pw.println(i);
  pw.println(d);
  pw.close();
}
```

File I/O Using Character Streams

- Use a **FileReader/FileWriter** to read/write characters from/to a file.

```
void copyTextFile(String fin, String fout)
    throws IOException {
  BufferedReader br = new BufferedReader(
                                new FileReader(fin));
  FileWriter fw = new FileWriter(fout);
  String line;
  while((line = br.readLine()) != null)
    fw.write(line + "\r\n");
  fw.close(); br.close();
}
```

The File Class

- The **File** class is useful for getting information about a file.
- For example:
  - It can tell you whether the file exists, is readable, is writeable, is hidden, and is a directory.
  - It can give you the file name, the parent directory's name, the time last modified, and the file length.
  - If the file is a directory, it can give you a list of the files in the directory.

Converting Numeric Strings

- One way to convert a number that was read as a string from the keyboard or from a text file into the proper numerical format is to use the **parse*XXX*( )** methods in the classes **Double, Float, Long, Integer, Short,** and **Byte**.
- Examples:

```
int x = Integer.parseInt("123");
double y = Double.parseDouble("1.23");
byte z = Byte.parseByte("12");
```

Chapter 12

Multithreaded Programming

Multithreading Fundamentals

- *Process-based* multitasking allows you to run two or more separate programs concurrently.
- *Thread-based* multitasking (multithreading) allows a single program to perform two or more tasks at once.
- Multithreading allows you to use the idle time present in one task (e.g., when waiting to send or receive information) to execute another task.
- It works with single-processor systems as well as multiprocessor/multicore systems.

Thread Class and **Runnable** Interface

- All processes have at least one thread: the main thread that is executed when your program begins.
- To create a second thread, your program will need to create an instance of the **Thread** class or a subclass of **Thread** that you define.
- Then you call the **start( )** method of the **Thread** class to start executing the new thread.

Specifying the Code to Run

- Approach 1: Create a subclass of **Thread** and put the code you wish to run in the method

  public void run( )

- Approach 2: Create an object of a class that implements the **Runnable** interface, which has one method

  public void run( )

  and pass that object to the **Thread**'s constructor.

Example of Approach 1

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Running");
  }
}

class TestThread {
  public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
  }
}
```

Example of Approach 2

```
class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Running");
  }
}

class TestRunnable {
  public static void main(String[] args) {
    Thread thread = new Thread(new MyRunnable());
    thread.start();
  }
}
```

Some **Thread** Methods

- To cause the thread to suspend for a given number of milliseconds:

  static void sleep(long *millis*)
- To get and set the priority of the thread:

  void setPriority(int *p*), int getPriority( )
- To determine whether the thread's **`run( )`** method has finished executing:

  boolean isAlive( )
- To wait until the thread on which it is called terminates:

  void join( )

Synchronization

- Synchronization of two threads is used to coordinate their activity.
- If you specify that a method or block of code is **synchronized**, only one thread can enter the method or code block at a time.
- Access is controlled by a *monitor* associated with an object.
- Whatever thread owns the monitor is allowed access.

Two Ways to Synchronize

```
// synchronize a method
synchronized void incrementAll(int[] data){
  for(int i = 0; i < data.length; i++)
    data[i]++;
}
// synchronize a block
void incrementAll(int[] data) {
  synchronized(data) {
    for(int i = 0; i < data.length; i++)
      data[i]++;
  }
}
```

Thread Communication

- All objects inherit **wait( ), notify( ),** and **notifyAll( )** from the **Object** class.
- These methods can only be called from within a synchronized context.
- Calling **wait( )** causes the thread to sleep and release the monitor, allowing other threads to get the monitor and execute the synchronized code.
- Calling **notify( )** or **notifyAll( )** awakens one or more threads waiting on the monitor.

Suspend, Resume, and Stop Threads

- To suspend, resume, or stop a thread, the usual approach is to create two boolean flag variables, one for suspend and resume and one for stop.
- The **run( )** method should periodically check the state of these variables.
- If the stop flag is true, the **run( )** method should exit.
- If the suspend flag is set to true, **run( )** should call **wait( )**. When the suspend flag is set to false, **run( )** should call **notify( )**.

Chapter 13

Enumerations, Autoboxing, and Annotations

Enumerations

- In its simplest form, an *enumeration* is a list of named constants that define a new data type.
- Example:

```
enum Transport {
   CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

- This example defines a new **Transport** class with 5 public static members, each referring to an object of that class.

Uses of Enumeration

```
enum Transport {
  CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

Transport tr = Transport.TRUCK;

if(tr == Transport.BOAT) ...

switch(tr) {
  case CAR: ...
  case TRUCK: ...
  case AIRPLANE: ...
}
```

Enumeration Methods

- All enumerations automatically have two predefined methods: **values( )** and **valueOf( )**:

  public static *enumType*[ ] values( )

  public static *enumType* valueOf(String *str*)

- Examples:

```
Transport[] allTransports = Transport.values();
for(Transport t : allTransports)
  System.out.println(t);

Transport tp = Transport.valueOf("AIRPLANE");
```

Enumerations Are Class Types

- You can add constructors, instance variables, and methods to enumerations.
- Example:

```
enum Transport {
  CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70),
  BOAT(22);

  private int speed;

  Transport(int s) { speed = s; }

  int getSpeed() { return speed; }
}
```

Enumerations Inherit from **Enum**

- All enumerations are subclasses of **Enum** and so inherit its methods, two of which are **ordinal( )** and **compareTo( )**.
- The **ordinal( )** method takes no parameters and returns the value's position in the list of values.
- The **compareTo( )** method takes another value as parameter and returns a negative, zero, or positive integer if this value's position is before, equal to, or after the other value's position.

Autoboxing

- The primitive types are not part of the object hierarchy, and they do not inherit **Object**.
- But there are times when you need an object representation of a primitive value.
- The type wrapper classes **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean** encapsulate a primitive value within an object.

Wrapper Methods and Constructors

- All type wrapper classes include constructors for wrapping a primitive value inside an object, e.g.,
  Integer(int *num*)
  Integer(String *str*)
- All type wrapper classes inherit methods for retrieving the primitive value, e.g.,
  int intValue( )
  double doubleValue( )
- The **toString( )** method returns the primitive value as a string.

Autoboxing Fundamentals

- *Autoboxing* is the process by which a primitive type is automatically boxed into its type wrapper when necessary.
- *Auto-unboxing* is the reverse process.
- Autoboxing automatically occurs whenever a primitive type must be converted into an object, and auto-unboxing takes place whenever an object must be converted into a primitive type.

Boxing and Unboxing

```
Integer obj = new Integer(100);  // manual boxing
int i = obj.intValue();          // manual unboxing

Integer intObj = 100; // autoboxing
int i = intObj;       // auto-unboxing

intObj.equals(3);   // autoboxing of 3
int x = 4 + intObj; // auto-unboxing of intObj
intObj++;  // auto-unbox, increment, and autobox
```

Annotations (Metadata)

- Annotations allow you to embed supplemental information into a source file.
- Some annotations have parameters, and others are just marker annotations.
- There are 8 general purpose built-in annotations: **@Retention, @Documented, @Target, @Inherited, @Override, @Deprecated, @SafeVarargs, @SuppressWarnings**.

Example of Marker Annotations

```
// class is deprecated
@Deprecated
class OldClass {

  // override the inherited equals() method
  @override
  public boolean equals(Object o) { return false; }
}
```

Chapter 14

Generics

Generics (Parameterized Types)

- Parameterized types enable you to create classes, interfaces, and methods in which the type of data on which they operate is specified as a parameter.
- In old code, many classes and methods used **Object** as the type of data they operated on.
- The disadvantage of the old approach is that explicit conversion with typecasting is often necessary to convert to the actual type of the data.

Example Without Generics

```
class KVPair{
  Object key, value;

  Object getKey() { return key; }
  Object getValue() { return value; }
  void setKey(Object ob) { key = ob; }
  void setValue(Object ob) { value = ob; }
}

Usage: KVPair pair = new KVPair();
       pair.setValue("Address");
       String v = (String) pair.getValue(); // typecast
```

Example with Generics

```
class KVPair<T>{
  T key, value;

  T getKey() { return key; }
  T getValue() { return value; }
  void setKey(T ob) { key = ob; }
  void setValue(T ob) { value = ob; }
}

Usage: KVPair<String> pair = new KVPair<String>();
       pair.setValue("Address");
       String v = pair.getValue(); // no typecast
```

Example Explained

- The **<T>** syntax indicates that **T** is a type parameter.
- **T** is a placeholder for the actual type, which is provided when a **KVPair** object is created.
- Use of generics allows us to avoid the type casting.
- That is, generics makes type casts automatic and implicit, and therefore add type safety.

Gotchas Using Generics

- You can only use class types for type parameters.
- For example, the following code is illegal:

```
KVPair<int> pair = new KVPair<int>();
```

- Generic types differ if they have different type arguments.
- For example, the last line of code is illegal:

```
KVPair<Integer> pair1 = new KVPair<Integer>();//OK
pair1.setKey(3); // OK autoboxing is performed
KVPair<String> pair2 = new KVPair<String>(); // OK
pair2.setKey("hi"); // OK
pair1 = pair2; // illegal
```

Generics with 2 Type Parameters

- You can use two or more type parameters:

```
class KVPair<K,V>{
  K key;
  V value;

  K getKey() { return key; }
  V getValue() { return value; }
  void setKey(K ob) { key = ob; }
  void setValue(V ob) { value = ob; }
}
```

USAGE:
```
KVPair<String,Integer> pair =
                    new KVPair<String,Integer>();
```

Bounded Types

- You can restrict some of the type parameters so that they must be a subtype or a supertype of a given type:

```
class KVPair<K extends Number, V super Integer>{
  K key;
  V value;

  K getKey() { return key; }
  V getValue() { return value; }
  void setKey(K ob) { key = ob; }
  void setValue(V ob) { value = ob; }
}
```

Bounded Types Explained

- The notation **T extends Number** means that **T** must be a **Number** or a subclass of **Number**.
- The notation **V super Integer** means that **V** must be **Integer** or a superclass of **Integer**.
- Other options:
  - **<T, K extends T>** means that the second type parameter must be the same as the first parameter or a subclass of the first parameter.
  - The wildcard **?** can match any type.

Generic Methods

- It is possible to have generic methods separate from generic classes (i.e., in non-generic classes).
- Example:

```
<T extends Number> long[] convertToLongs(T[] data)
{
  long[] result = new long[data.length];
  for(int i = 0; i < data.length; i++)
    result[i] =  data[i].longValue();
  return result;
}
```

More on Generic Methods

- Note that generic methods, when called, do not need to specify the type arguments.
- Instead, the compiler deduces the type arguments from the types of the arguments to the method call.
- Constructors can be generic-like methods.

Inheritance and Generics

- Generic classes can be extended by generic subclasses.
- The subclasses must pass type arguments up to the superclass that needs them.
- Example:

```
class Sub<T,V> extends Sup<T> {
   ...
}
```

Interfaces and Generics

- Interfaces can also be generic.
- Any class that implements a generic interface must also be generic unless it implements a specific type of the interface.
- Examples:

```
class C<T> implements I<T> { ... }
class D implements I<String> { ... }
```

Raw Types and Legacy Code

- For backwards compatibility with legacy pre-generic code, all generic classes can be used without any type arguments.
- In that case, the class is used with **Object** as the value of all type parameters.
- Example using the **KVPair<K,V>** class:

```
KVPair pair = new KVPair(); // no type arguments
pair.setKey("hi");
String s = (String) pair.getkey(); // cast needed
```

Generic Restrictions

- Static variables and methods cannot use the type parameters declared by their generic class (but you can have static generic methods).
- Example:

```
class C<T> {
  static T x; // illegal
  static T getX() { return x; } // illegal
  static <W> void setX(W w) {  // legal
    ...
  }
}
```

More Generic Restrictions

- You can't create an instance of a type parameter.
- You can't create an array whose element type is the type parameter nor an array of type-specific generic references.
- Example:

```
class C<T> {
  T x = new T(); // illegal
  T[] x = new T[10]; // illegal
  C<String>[] data= new C<String>[10]; // illegal
  C<?>[] data = new C<?>[10]; // legal
}
```

Chapter 15

Applets and the Remaining Java Keywords

Applet Basics

- Applets are small programs designed for transmission over the Internet and run within a browser.
- There are two varieties of applets: AWT-based and Swing-based.
- In this chapter we look only at AWT-based applets.

Example

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Yay, Java!", 20, 20);
  }
}
```

Executing an Applet

- Compile the applet as usual.
- Run the applet by using an applet viewer tool or by embedding it in a web page and using a web browser.
- This HTML executes **SimpleApplet**:

```
<applet code="SimpleApplet"
        width=200 height=60>
</applet>
```

Applet Lifecycle Methods

```
// Called first.
public void init() {...}

// Called second, after init(). Also called
// whenever the applet is restarted.
public void start() {...}

// Called when the applet is stopped.
public void stop() {...}

// Called when the applet is terminated.
public void destroy() {...}
```

More on Applet Methods

- The **Applet** class provides default empty implementations of these methods.
- An applet waits for the runtime system to call one of its methods.
- In the method body, the applet must take appropriate action and then quickly return control to the system.
- A repetitive task requires a separate thread.
- When an applet needs to update the information it displays, it should call **repaint( )**.

Using the Status Window

- An applet can output a message to the status window of the browser on which it is running.
- To do so use the method:

  void showStatus(String *msg*)
- Example:

```
showStatus("This appears in the status window");
```

Passing Parameters to Applets

- To pass a parameter, put its name and value in a PARAM attribute of the APPLET tag.
- Example:

```
<applet code="MyApplet" width="100" height="100">
  <param name="version" value="2">
</applet>
```

- To retrieve the parameter, use the method:

  String getParameter(String *paramName*)

- Example:

```
String version = getParameter("version");
```

The **instanceof** Keyword

- The **instanceof** operator is used in an expression as follows:
  objectref instanceof *type*
- If the object referenced is of the specified type or can be cast into the type, then the expression evaluates to true. Otherwise it is false.
- Example:

```
if("hi" instanceof String)
  System.out.println("Yes"); // "Yes" is printed
```

The **assert** Keyword

- **assert** statement—asserts that a condition is expected to be true at a point during execution.
- If the asserted condition is true, no special action takes place. If the condition is false, an **AssertionError** is thrown.
- **assert**s are not usually used for released code.
- Example:

```
assert x > 0;
```

- To enable assertion checking at runtime, use

```
java –ea className
```

The Remaining Java Keywords

- **volatile** modifier—tells the compiler that a variable can be changed unexpectedly by another thread.
- **transient** modifier—indicates that an instance variable is not part of the persistent state of an object.
- **strictfp** modifier—requires that Java's floating point calculations strictly follow the IEEE 754 standard.
- **native** modifier—used to declare native code methods (methods written in other languages and compiled into executable code).

Chapter 16

Introduction to Object-Oriented Design

Why Design Matters

- Designing software that is intended for long-term, heavy use requires a considerable investment of time and energy.
- In such large systems, bugs are inevitable.
- Therefore, we want to design the software to minimize the initial number of bugs and to make it easy to modify the software without introducing additional bugs.

Properties of Elegant Software

- **Usable**—It is easy for the client to use.
- **Complete**—It satisfies all the client's needs.
- **Robust**—It deals with unusual situations gracefully.
- **Efficient**—It uses a reasonable amount of time and other resources.
- **Scalable**—It will perform correctly and efficiently even when the problem grows in size by several orders of magnitude.

More Properties

- **Readable**—It is easy for a software engineer to understand the design and the code.
- **Reusable**—It can be reused in other settings.
- **Simple**—It is not unnecessarily complex.
- **Maintainable**—Defects can be found and fixed easily without introducing new defects.
- **Extensible**—It can easily be enhanced without breaking existing code.

Naming Conventions

- *Use intention-revealing names.*
- Names for **void** methods should say what the method does.
- Names for functions should say what the method returns.
- Examples:

```
void setName(String newName)
int getAge()
int length()
```

More Naming Conventions

- Names for **void** methods are typically verbs or verb phrases.
- Class names are typically nouns.
- Interface names are typically adjectives ending in "-able" or "-ible".
- Variable names are typically nouns indicating the value and the role of the variable.

Method Cohesion

- *Methods should do one thing only and do it well.*
- Methods should have one cohesive action.
- Typically, a method shouldn't calculate a value, display it, *and* save it to a file.
- In case you may want only one of those three actions done, break the method into three separate methods.

Well-formed Objects

- *A non-private method should keep an object in a well-formed state.*
- A well-formed state is usually specified with class invariants.
- A *class invariant* is a statement giving requirements about the state of objects between public method calls.
- Before a non-private method returns, it should ensure that all class invariants are satisfied.

Example of a Class Invariant

- Suppose a class stores an array of items.
- One possible class invariant is that the array length must match the number of items stored.
- That is, the array must have no unused slots.

Documentation

- *Include complete external documentation for the user and complete internal documentation for the developer.*
- *Internal documentation* is documentation for developers looking at the source code.
- Such documentation should provide information not readily available from the code itself.
- It should summarize what is being done, why it is being done, and why it is being done this particular way.

More on Internal Documentation

- Internal documentation should also clearly state any class or method invariants that exist.
- It should summarize the intent or purpose of the code.
- Internal documentation mostly consists of comments in the source code.
- Self-documenting code is rarely possible, so such documentation is needed.

External Documentation

- *External documentation* is for users of the code who can't look at or don't care about the source code itself.
- It describes the public classes, interfaces, methods, fields, and packages and how to use them.
- Strive for complete documentation for all methods.
- Use Javadoc notation when possible.

External Documentation Example

```
/**
 * Returns the n-th root of the double value.
 * If either n < 0 or n is even and value < 0,
 * then an IllegalArgumentException is thrown.
 *
 * @param value the double whose root is desired
 * @param n the integer indicating the root to compute
 * @return the n-th root of the value
 *    If n is even, the positive n-th root is returned.
 * @throws IllegalArgumentException
 *    if either n < 0 or n is even and value < 0.
 */
public double nthRoot(double value, int n)
```

Class Cohesion

- *Classes, like methods, should do one thing only and do it well.*
- A class should model one concept, and all the methods in the class should be related to and appropriate for that concept.
- Avoid "god" classes that do all the work for other classes.

Expert Pattern

- *The object that contains the necessary data to perform a task should be the object that performs the task.*
- Example: To find an object in a collection, you could search the collection, or the collection could do the searching for you.
- The Expert pattern says the collection should do the searching for you.

Duplication

- *Avoid duplication of*
  - *data*
  - *code*
  - *processes*
  - *documentation*
- Duplication is wasteful and can lead to errors.
- Each set of data should have one "gatekeeper" class that maintains it.

Complete Interface

- *Give a class a complete interface.*
- Don't leave out methods that others may need.
- Example: If a GUI component is selectable, add methods
  - boolean isSelected( )
  - void setSelected(boolean *on*)

Change Happens

- *Design your classes so that they can handle change.*
- Change occurs when you are:
  - removing bugs
  - modifying/maintaining code
  - enhancing code

Open-Closed Principle

- *Design software so that it is easy to extend by adding new classes, extending existing classes, and reusing existing classes rather than by modifying existing classes.*
- Minimize the modification of existing, working code.
- Instead, extend the code by adding new classes that incorporate the desired changes.

Use Interfaces

- *Code to interfaces, not classes.*
- Give variables the widest type feasible.
- Example:

```
AcmeClock obj;  // limited values
ElectricClock obj;  // wider range of values
ElectricAppliance obj; // even wider range
```

Encapsulation and Information Hiding

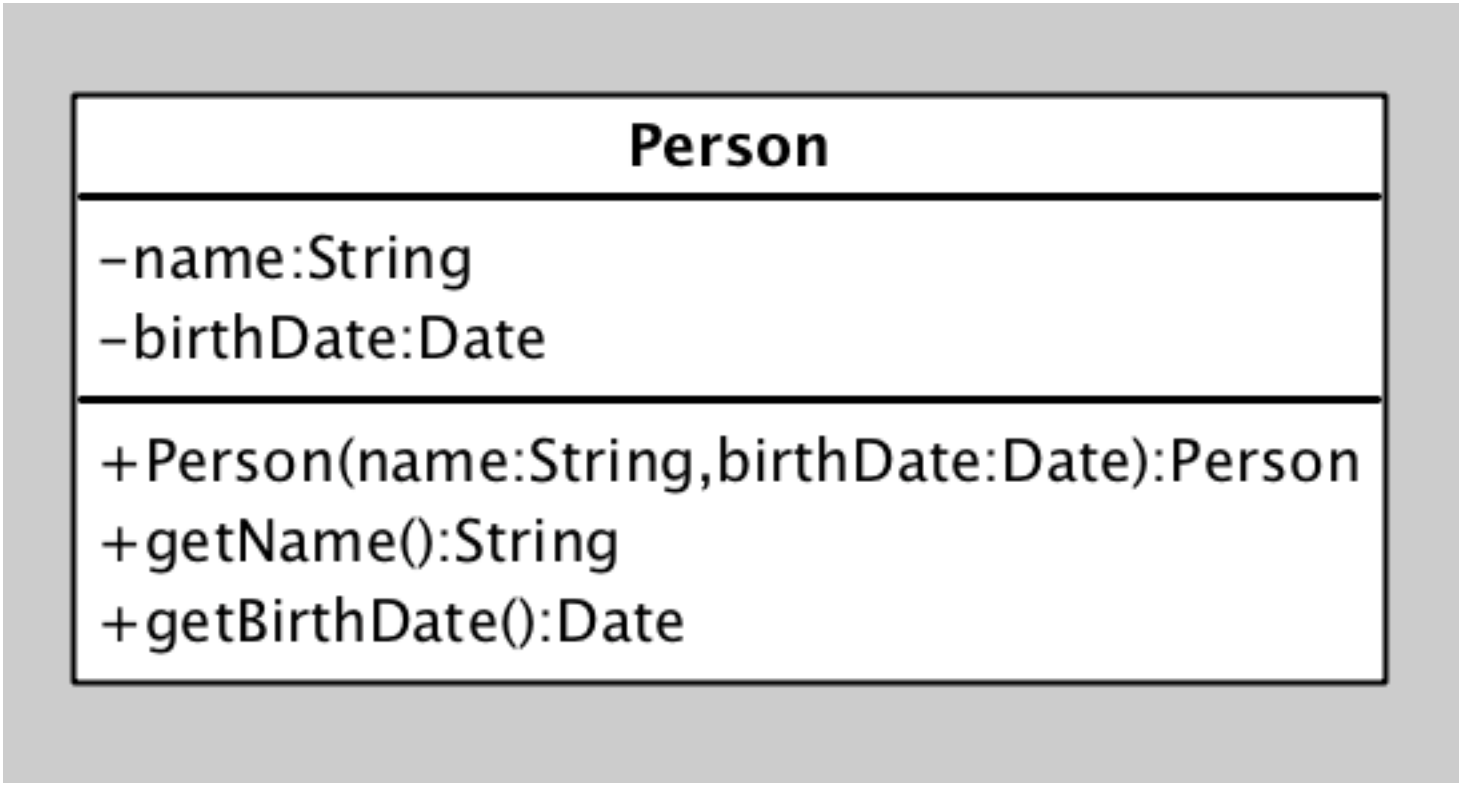
- *Use encapsulation*—group together related items and put a wall around them.
- Classes and packages provide opportunities for encapsulation.
- *Use information hiding* as much as possible.
- One way to hide information is to keep instance variables private.

Law of Demeter

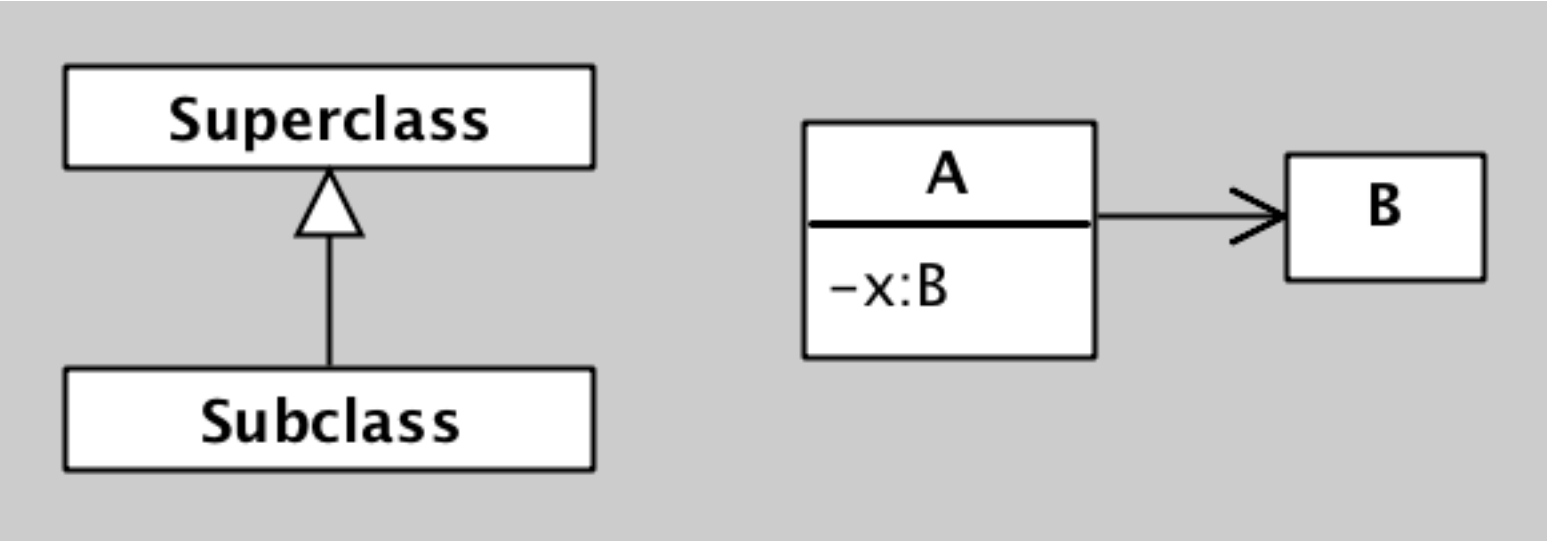
- *Invoke methods only on objects that you either create yourself or have direct access to.*
- Don't talk to strangers, for example, objects returned from other method calls.
- Example that fails the Law of Demeter:

```
Balance balance =
                centralControl.getBank(b).
                    getBranch(r).getCustomer(c).
                        getAccount(a).getBalance();
```

UML Class Diagrams



UML Inheritance and Associations



Proper Use of Inheritance, Part 1

- *Code reuse* perspective: If class **A** and class **B** have some identical methods or data, make one a subclass of the other to avoid duplication.
- Problem: Both a **Person** and a **Dog** class might have a common **name** property and a common **getName( )** method. Which should be the subclass?

Proper Use of Inheritance, Part 2

- *Is-a* perspective: If an object of class **A** "is an" object of class **B**, then class **A** should be a subclass of **B**.
- Problem: A square "is a" rectangle, but if the **Rectangle** class has a **setSides(int *newWidth*, int *newHeight*)** method, should a **Square** class inherit it?
- If these objects are immutable, then inheritance is more appropriate.

Principle of Least Astonishment

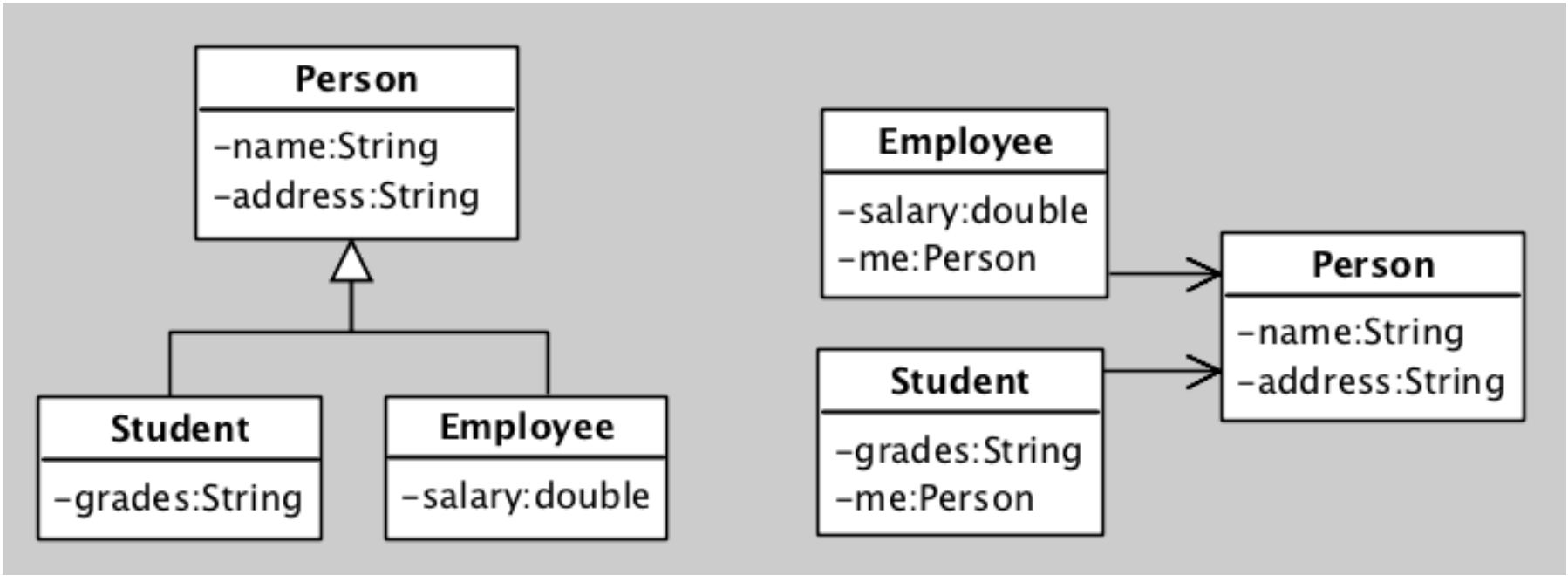
- *If a client thinks she has a reference to an object of type A but actually has a reference to an object of subtype B, there should be no surprises when she invokes methods on the object.*

Proper Use of Inheritance, Part 3a

- *Similar behavior* perspective: If classes **A** and **B** have similar interfaces with similar behavior, then one should be a subclass of the other.
- Example: A **Person** class whose methods are a subset of the methods of a **Student** class.
- In that case, it seems logical to make **Student** a subclass of **Person**.
- Problem: What happens when the student graduates?

Proper Use of Inheritance, Part 3b

- Do not use inheritance if one class is just a temporary role played by the other class.
- Instead, use delegation.



Proper Use of Inheritance, Part 4

- *Polymorphism* perspective: If we want to assign an object of class **A** to a variable of class **B**, then **A** should be a subclass of **B**.
- In this case, inheritance is probably appropriate.
- Example: In the Java AWT framework, a **Container** has a method

  Component add(Component *comp*)
- Any object you want to add to a **Container** must be of a subclass of **Component**.

Costs of Inheritance

- Code for a method low in an inheritance tree may be spread out among all ancestors.
- Subclasses are tightly coupled to their superclasses.

Design Patterns

- *Design patterns* are some of the software industry's best practices for solving problems within certain constraints.
- Patterns have names to facilitate discussion of designs.

Adapter Pattern

To connect two systems with different interfaces, do not change the systems' interfaces—instead, create an "adapter" class that connects the two systems.

Adapter Pattern Example

- Example: A class **A** provides printing services to any class implementing an interface **I** and you have a class **B** you want printed but it doesn't implement **I**.
- Instead of modifying **A** or **B**, create a class **C** that implements **I** and has a reference to class **B**.
- Have class **C** extract the needed information from **B** and pass it to class **A** to be printed.

Observer Pattern

- Useful in situations in which many objects need to be aware of changes to a particular subject.
- Example: Classes that handle fees or fraud need to be aware of changes to balances in savings accounts.
- Observer pattern: The subject keeps a list of observers and notifies them whenever a change occurs.

Observer Pattern (continued)

- Avoids the need for the observing classes to repeatedly poll the subject for any changes.
- Observers can register/unregister with the subject dynamically.
- Also called "subscribers" and "publishers" instead of observers and subjects.

Chapter 17

Swing Fundamentals

Swing Fundamentals

- Swing is a collection of classes and interfaces that offer a rich set of visual components.
- Swing components are *lightweight*—they are written entirely in Java.
- As such, they can have a variety of look-and-feels.

Model-View-Controller (MVC)

- *model*—corresponds to state information of a component
- *view*—determines how the component is displayed on the screen
- *controller*—determines how the component reacts to the user
- Swing uses a modified version of MVC that combines the view and controller.

Components

- Swing consists of two key items: components and containers.
- The **JComponent** class is the superclass of all but four Swing components.
- All Swing components are in the **javax.swing** package.
- There are currently 46 Swing components, including **JButton**, **JDialog**, **JCheckBox**, **JMenu**, and **JMenuItem**.

Containers

- Top-level containers in Swing: **JFrame, JApplet, JWindow, JDialog**.
- These do not inherit from **JComponent**.
- Top-level containers cannot be contained in another container.
- Top-level containers contain several components called *panes*.
- The *content pane* holds the components the user interacts with.

Layout Managers

- Layout managers control the position of components within a container.
- Layout managers implement the **LayoutManager** interface.
- Two of the layout managers are:
  - **FlowLayout**—A simple layout that positions components left-to-right, top-to-bottom.
  - **BorderLayout**—Positions components within the center or on the borders of the container. This is the default layout for a content pane.

Simple Swing Demo

```
import javax.swing.*;
class SwingDemo {
  SwingDemo() {
    JFrame jfrm = new JFrame("A Simple Swing App");
    jfrm.setSize(275, 100);
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel jlab = new JLabel(" Swing: the Java GUI.");
    jfrm.add(jlab);
    jfrm.setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {  new SwingDemo(); }
    });
  }
}
```

Event Dispatching Thread

- Swing applications are event driven.
- Swing provides an event-dispatching thread to handle events, separate from the main thread.
- All Swing GUI components must be created and updated on the event-dispatching thread.
- In the **SwingDemo** class, the call to **SwingUtilities.invokeLater( )** causes the **Runnable** code to be executed on the event-dispatching thread.

Event Handling

- Event handling is a large part of any Swing-based application.
- Events are generated when, for example, the user clicks a button, types a key, or selects an item from a list.
- In Swing, a source generates an event and sends it to one or more listeners.
- Listeners just wait until they receive events, and then they process the events.

Event Handling (continued)

- This approach is called the *delegation event model* since the GUI element delegates the processing of the event to a separate object.
- All listeners must register with the source if they wish to be notified of events.
- This is essentially the Observer pattern described in Chapter 16.

Event Sources

- Event sources have methods of the form:
  public void add*Type*Listener(*Type*Listener *el*)
  public void remove*Type*Listener(*Type*Listener *el*)
- These methods are called to add or remove listeners from the notification list.
- Example:

```
JButton b = new JButton();
ActionListener el = new ActionListener(){ ... }
b.addActionListener(el);
```

Event Listeners

- Listeners are notified when events occur.
- They have two requirements:
  - They must register with a source.
  - They must implement a method to receive and process the event.
- The methods they implement are defined by interfaces.
- The methods must quickly process the event and return or else use a separate thread, to avoid tying up the event-dispatching thread.

Some Event Classes

Event Class	Description	Event Listener
ActionEvent	Generated when an action occurs within a control, such as when a button is pressed.	ActionListener
FocusEvent	Generated when a component gains or loses focus.	FocusListener
ItemEvent	Generated when an item is selected, such as when a check box is clicked.	ItemListener
KeyEvent	Generated when input is received from the keyboard.	KeyListener
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.	MouseListener and MouseMotionListener

Adapter Classes

- Adapter classes provide empty implementations of event listener interface methods.
- You can extend one and implement only those methods in which you are interested.
- Sample adapter classes:

Adapter Class	Implements
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener

Using a Push Button

- A push button is an instance of **JButton**.
- A push button generates an **ActionEvent**.
- The associated listener interface is **ActionListener**, which has one method:
  public void actionPerformed(ActionEvent *ae*)
- In summary, to handle a button press event,
  - Create a class that implements **ActionListener**.
  - Register the class with the **JButton** by calling the button's **addActionListener( )** method.

Using a Push Button (continued)

- When the user presses the **JButton**, it generates an **ActionEvent**.
- All registered listeners are notified; more precisely their **actionPerformed( )** methods are called.
- The **ActionEvent** is passed as the argument to the **actionPerformed( )** method.
- **ActionEvent** class has several methods to give you more information about the event.

Push Button Example

```
class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent ae) {
    label.setText(ae.getActionCommand());
  }
}
...
JButton button = new JButton("First");
ButtonHandler handler = new ButtonHandler();
button.addActionListener(handler);
...

// when the button is pushed, the word "First" is
// displayed as the text in a label.
```

Introducing JTextFields

- A **JTextField** control enables the user to enter a line of text.
- If the user presses ENTER when inputting into a text field, an **ActionEvent** is generated.
- Listeners must implement the **ActionListener** interface.

Introducing JTextFields (continued)

- One of its constructors:
  JTextField(int *cols*)
- It creates a text field with the given width.
- The **JTextField** class has methods for accessing the text that the field displays:
  - void setText(String *text*)
  - String getText( )

JTextField Example

```
class TextFieldHandler implements ActionListener {
  public void actionPerformed(ActionEvent ae) {
    label.setText(ae.getActionCommand());
  }
}
...
JTextField field = new JTextField(10);
TextFieldHandler handler = new TextFieldHandler();
field.addActionListener(handler);
...

// when ENTER is pressed, the contents of the text
// field are displayed in a label.
```

Using Anonymous Inner Classes

- Here are some ways to handle events:
  - Have the class with the main method be the listener.
  - Create a separate listener class.
  - Use an anonymous inner class as the listener.
- Example of the third way:

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    label.setText(ae.getActionCommand());
  }
});
```

Chapter 18

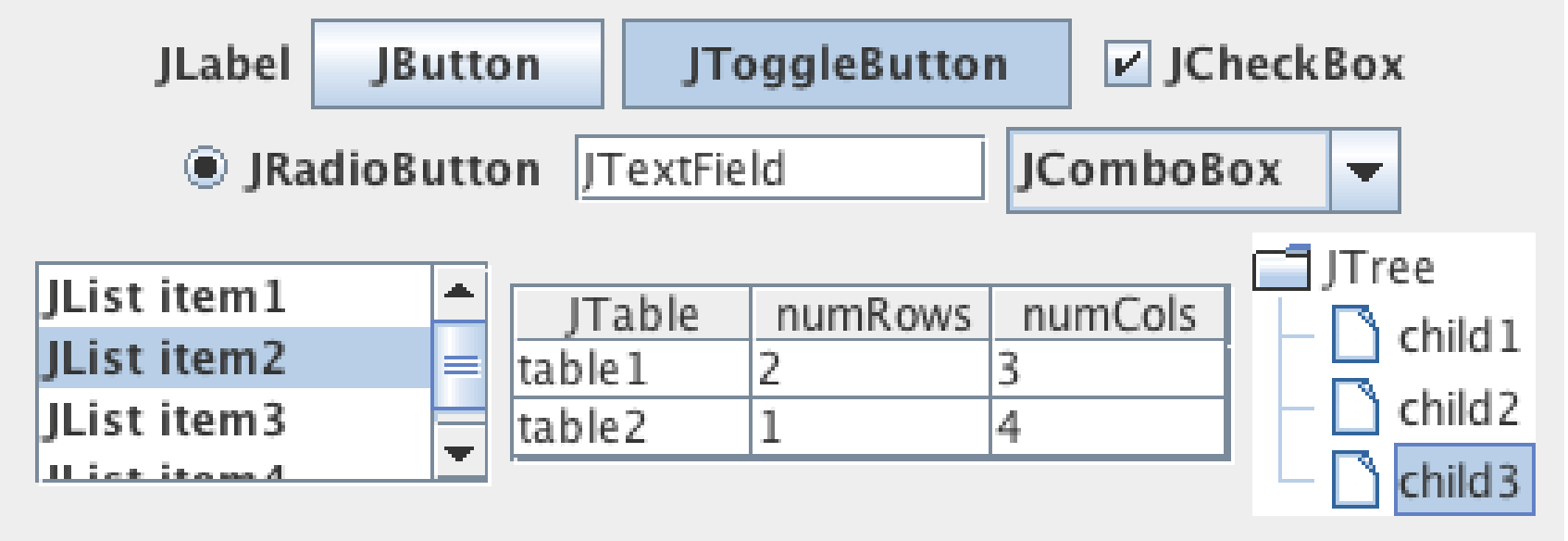
Exploring Swing Controls

Swing Components

- In this chapter, we will discuss the following controls, as well as the **ButtonGroup** and **ImageIcon** classes.

JButton	JCheckBox	JComboBox
JLabel	JList	JRadioButton
JScrollPane	JTable	JTextField
JToggleButton	JTree	

Visual Display of Some Swing Controls



JLabel and ImageIcon

- **JLabel** displays a label and/or an icon.
- It does not respond to user input.
- Two of **JLabel**'s constructors:
  JLabel(String *str*)
  JLabel(Icon *icon*)
- **Icon** interface is implemented by **ImageIcon**.
- One of **ImageIcon**'s constructors:
  ImageIcon(String filename)
- The file can be of type **gif**, **jpeg**, or **png**.

Swing Buttons

- The following buttons are subclasses of **AbstractButton**:

JButton	A standard push button
JToggleButton	A two-state (on/off) button
JCheckBox	A standard check box
JRadioButton	A group of mutually exclusive check boxes

Buttons and ActionEvents

- When clicked, a button generates an **ActionEvent**.
- The listener can determine the component that generated the event from the **ActionEvent**, using two **ActionEvent** methods:

  String getActionCommand( )

  Object getSource( )
- The component sets the action command with:

  void setActionCommand(String *cmd*)

Handling ItemEvents

- An **ItemEvent** is generated when an item, such as a toggle button, check box, or radio button, is selected.
- To handle an **ItemEvent**, a class must implement the **ItemListener** interface, which has one method:

  void itemStateChanged(ItemEvent *ie*)
- To find the component that generated the event, you can use the **ItemEvent** method:

  Object getItem( )

Handling ItemEvents (continued)

- When an item event occurs, the component will be in one of two states: selected or deselected.
- To get the new state, call **ItemEvent** method:
  int getStateChange( )
- The value returned will be **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**.
- You can also use the **AbstractButton** method:
  boolean isSelected( )

JButton and JToggleButton

- A **JButton** represents a push button.
- When pressed, a **JButton** generates an **ActionEvent**.
- A **JToggleButton** represents a toggle button that looks like a push button, but it stays pressed rather than released and needs to be pressed again to release it.
- **JToggleButton** generates both an **ActionEvent** and an **ItemEvent** when pressed.

Check Boxes

- A **JCheckBox** represents a check box.
- It is a subclass of **JToggleButton**.
- When selected or deselected, an **ItemEvent** is generated.
- One of its constructors:
  JCheckBox(String *str*)
- It creates a check box that has the text specified by *str* as a label.

Radio Buttons

- **JRadioButton**s represents radio buttons, which are groups of mutually exclusive buttons, only one of which can be selected at any one time.
- A button group is created using the **ButtonGroup** class.
- Add **JRadioButtons** to a button group by using the **ButtonGroup**'s method:

  void add(AbstractButton *btn*)
- When clicked, a **JRadioButton** generates an action event.

JTextField

- **JTextField**s allow the user to enter a line of text.
- An **ActionEvent** is generated when ENTER is pressed.
- Useful methods in the **JTextField** class:

  String getText( )
  void setText(String *text*)
  String getSelectedText( )
  void cut( )
  void copy( )
  void paste( )

JScrollPane

- **JScrollPane** is a lightweight container that automatically handles the scrolling of another lightweight component.
- Scroll bars automatically appear if the object being viewed is too large for the viewable area.
- One of its constructors:

  JScrollPane(Component *comp*)
- The component to be scrolled is specified by *comp*.

JList

- **JList** supports the selection of one or more items from a list.
- Usually you wrap a **JList** in a **JScrollPane**.
- When the user makes a selection, a **ListSelectionEvent** is generated.
- It is handled by a **ListSelectionListener**, an interface with one method:

  void valueChanged(ListSelectionEvent *le*)

JList (continued)

- **JList** is generic in JDK7 and is declared here:
  class JList<E>
- One of its constructors:
  JList(E[ ] *items*)
- You can specify whether users can select only one item, a single range of items, or multiple ranges of items at a time.
- To get the index of the first selected item, use
  int getSelectedIndex( )

JComboBox

- Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class.
- Like **JList**, **JComboBox** is generic in JDK7.
- One of its constructors: JComboBox(E[ ] *items*)
- You can dynamically add and remove items.
- Like **JList**, **JComboBox** generates both an action event and an item event.
- To obtain the selected item use the method:

  Object getSelectedItem( )

JTree

- A **JTree** presents a hierarchical view of data, with collapsible and expandable subtrees.
- Two of its constructors:

  JTree(Object[ ] *obj*)

  JTree(TreeNode *tn*)
- A **JTree** generates several events, including **TreeSelectionEvent**s.
- To listen for this event, implement the interface **TreeSelectionListener**, with one method:

  void valueChanged(TreeSelectionEvent *te*)

Building a JTree

- One easy way to build a **JTree** is to create **DefaultMutableTreeNode** objects.
- One of its constructors:
  DefaultMutableTreeNode(Object *obj*)
- Then build a hierarchy using the **add( )** method of the **DefaultMutableTreeNode** class to add some nodes as children.

JTable

- A **JTable** is a component that displays rows and columns of data.
- You can resize and drag the columns.
- Each column has a header.
- One of its constructors:
  JTable(Object[ ][ ] *data*, Object[ ] *colHeads*)
- A **JTable** generates several events.

Other Swing Components

- Here are a few more controls you might find interesting:

JFormattedTextField	JPasswordField	JProgressBar
JScrollBar	JSlider	JSpinner
JTabbedPane	JTextArea	JToolBar

Chapter 19

Working with Menus

Menu Basics

- The Swing menu system uses these classes:

Menu Classes	Description
JMenuBar	An object that holds the top-level menu for the application.
JMenu	A standard menu. A menu consists of one or more **JMenuItem**s.
JMenuItem	An object that populates menus.
JCheckBoxMenuItem	A check box menu item.
JRadioButtonMenuItem	A radio button menu item.

Creating a Main Menu for a JFrame

- Create a **JMenuBar** object, which is a container for menus.
- Add instances of **JMenu**, which define menus.
- To each **JMenu**, add objects of class **JMenuItem**.
- Add the **JMenuBar** to the **JFrame** by calling the **JFrame** method:

  void setJMenuBar(JMenuBar *mbar*)

JMenuItem and Listeners

- **JMenuItem** is a subclass of **AbstractButton** (the superclass of **JButton** and other buttons), and so menu items are essentially buttons.
- When a menu item is selected, an **ActionEvent** is generated and so any handler must implement the **ActionListener** interface.
- Three subclasses of **JMenuItem** are
  - JCheckBoxMenuItem
  - JRadioButtonMenuItem
  - JMenu (for submenus)

Some Useful JMenuBar Methods

- JMenu add(JMenu *menu*)
- Component add(Component *menu*, int *idx*)
- void remove(Component *menu*)
- void remove(int *idx*)
- int getMenuCount( )

Useful JMenu Constructor and Methods

- JMenu(String *name*) // constructor
- JMenuItem add(JMenuItem *item*)
- JMenuItem add(Component *item*, int *idx*)
- void addSeparator( )
- void insertSeparator(int *idx*)
- void remove(JMenuItem *menu*)
- void remove(int *idx*)
- int getMenuComponentCount( )
- Component[ ] getMenuComponents( )

JMenuItems

- One of its constructors:

  JMenuItem(String *name*)
- One of its methods:

  void setEnabled(boolean *enable*)

Mnemonics and Accelerators

- Two kinds of keyboard shortcuts
- A mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed.
- An accelerator is a key that lets you select a menu item without having to activate the menu first.
- Example:

  CTRL-S is an accelerator for the "Save" menu item.

Adding Mnemonics

- Use the **JMenuItem** constructor or method:
  JMenuItem(String *name*, int *mnem*) //constructor
  void setMnemonic(int *mnem*)
- The value of *mnem* should be a **java.awt.event.KeyEvent** constant, such as
  - KeyEvent.VK_A
  - KeyEvent.VK_Z

Adding Accelerators

- Call the **JMenuItem** method

  void setAccelerator(KeyStroke *ks*)
- To create a **KeyStroke**, use the **KeyStroke** method:

  static KeyStroke getKeyStroke(int *ch*, int *modifier*)
- *modifier* should be one or more of:

InputEvent.ALT_DOWN_MASK	InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.CTRL_DOWN_MASK	InputEvent.META_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK	

Adding Images and Tooltips

- Other constructors that add icons:

  JMenuItem(Icon *image*)

  JMenuItem(String *name*, Icon *image*)
- A *tooltip* is a short piece of text that is automatically displayed when the mouse hovers over a component.
- To add a tooltip, use the **JComponent** method:

  void setToolTipText(String *tipStr*)

JCheckBoxMenuItem

- This item works like a stand-alone check box.
- It generates action events and item events.
- Two of its constructors:

  JCheckBoxMenuItem(String *name*)

  JCheckBoxMenuItem(String *name*, boolean *state*)
- If *state* is true, the check box is initially checked.
- Other constructors allow you to add icons.

JRadioButtonMenuItem

- This item works like a stand-alone radio button, generating item and action events.
- It must be put in a button group in order to exhibit mutually exclusive behavior.
- Two of its constructors:

  JRadioButtonMenuItem(String *name*)

  JRadioButtonMenuItem(String *name*,
  boolean *state*)

- Other constructors allow you to add icons.

Chapter 20

Dialogs

Dialogs

- A *dialog* is a separate window that requests some form of response from the user.
- It will contain at least a message and a button.
- We will look at 3 classes in Swing that support dialogs: **JDialog**, **JOptionPane**, and **JFileChooser**.
- Some dialogs are complicated, such as configuration or preference dialogs.
- Other dialogs simply prompt the user and wait for a response.

JOptionPane

- **JOptionPane** is a dialog class that supports four basic types of dialogs:
  - Message—simply displays a message
  - Confirmation—asks a question usually with a Yes/No answer and waits for a response
  - Input—asks the user to enter a string or select an item from a list
  - Option—specifies a custom list of options from which the user can choose

JOptionPane (continued)

- **JOptionPane** dialogs are *modal*—they demand a response before the program will continue.
- Nonmodal dialogs are also possible, using the **JDialog** class.
- You can use the **JOptionPane** constructors to create a dialog, but usually you use one of its 4 static factory methods:

showConfirmDialog( )	showInputDialog( )
showMessageDialog( )	showOptionDialog( )

showMessageDialog( )

- The **showMessageDialog( )** method creates a dialog that displays a message and waits until the user presses the OK button.
- One of its forms is shown here:

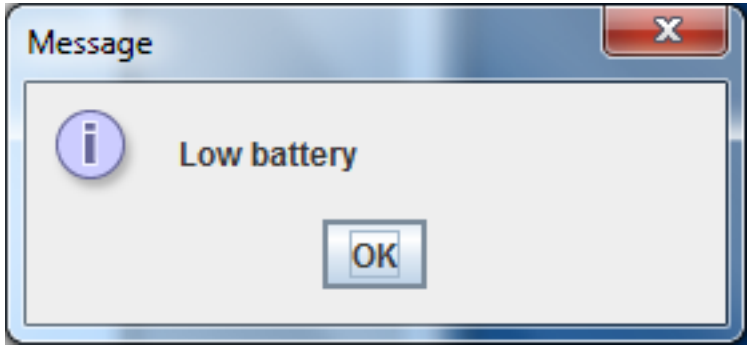
  static void showMessageDialog(Component *parent*, Object *msg*) throws HeadlessException

- The **HeadlessException** is thrown if the method is called in a noninteractive environment.

showMessageDialog( ) continued 1

- The *parent* is a component relative to which the dialog is displayed.
- If **null** is used as the value of *parent*, the dialog is typically centered on the screen.
- Example:

```
JOptionPane.showMessageDialog(null,"Low battery");
```



showMessageDialog( ) continued 2

- Another version with more options:

  static void showMessageDialog(Component *parent*,
  Object *msg*, String *title*, int *msgT*, Icon *image*)
  throws HeadlessException

- You can specify the window title, the icon to display, and one of these message types:

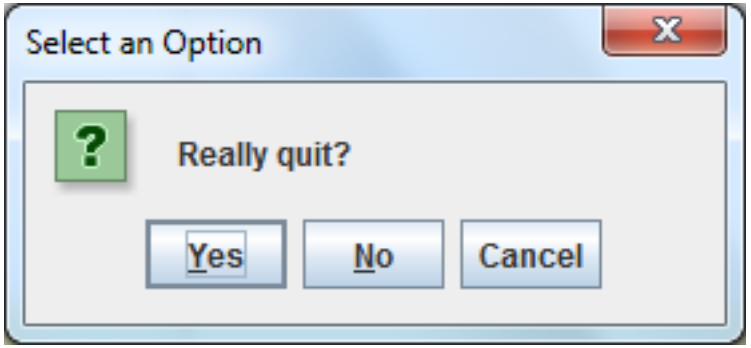
ERROR_MESSAGE	INFORMATION_MESSAGE
PLAIN_MESSAGE	WARNING_MESSAGE
QUESTION_MESSAGE	

showConfirmDialog( )

- Displays a dialog that requests a basic Yes/No response from the user.
- Simplest version:

  static int showConfirmDialog(Component *parent*,
  Object *msg*) throws HeadlessException

- Contains 3 buttons called Yes, No, and Cancel.



showConfirmDialog( ) continued 1

- The dialog returns one of these **JOptionPane** constants:

CANCEL_OPTION	Returned if the user clicks Cancel
CLOSED_OPTION	Returned if the user closes the dialog without making a choice
NO_OPTION	Returned if the user clicks No
YES_OPTION	Returned if the user clicks Yes

showConfirmDialog( ) continued 2

- Another version gives the user more options:

  static int showConfirmDialog(Component *parent*, Object *msg*, String *title*, int *optT*, int *msgT*, Icon *image*) throws HeadlessException

- *optT* should be a **JOptionPane** constant:

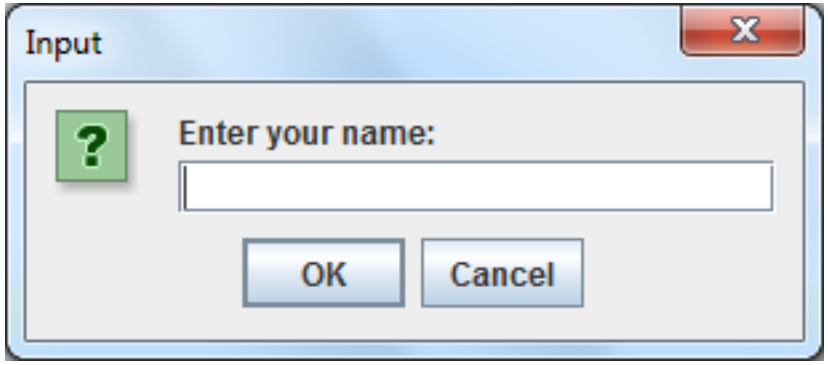
OK_CANCEL_OPTION	The dialog includes buttons for OK and Cancel.
YES_NO_OPTION	The dialog includes buttons for Yes and No.
YES_NO_CANCEL_OPTION	The dialog includes buttons for Yes, No, and Cancel.

showInputDialog( )

- The simplest form displays a text field into which a user can enter a string:

  static String showInputDialog(Object *msg*)
  throws HeadlessException

- If the user presses OK, the string entered by the user is returned.
- If the user presses Cancel, null is returned.



showOptionDialog( )

- This allows you to create a dialog tailored more precisely to your needs:

```
static int showOptionDialog(Component parent,
        Object msg, String title, int optT, int msgT,
        Icon image, Object[ ] options, Object initVal)
                throws HeadlessException
```

- The options (i.e., the buttons) from which the user can choose are specified by *optT*.
- If *options* is not null, the *optT* parameter is ignored.

showOptionDialog( ) continued

- The options are usually strings or icons, which become the labels of the buttons.
- The dialog returns the index of the button that was clicked.

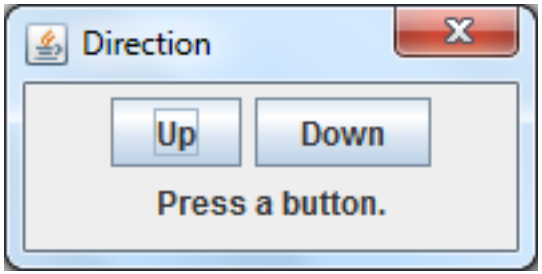
JDialog

- Use **JDialog** when you need more fields or special handling than **JOptionPane** provides.
- Allows you to create modal or modeless dialogs.
- **JDialog** is a top-level container.
- One of its constructors:

  JDialog(Frame *parent*, String *title*,
  boolean *isModal*)

Constructing a JDialog

- Create a **JDialog** much like you create a **JFrame**, including
  - adding components
  - setting a layout manager
  - creating a menu bar
  - setting its size
  - setting it visible



JFileChooser

- Swing provides a built-in dialog that handles the task of allowing the user to select a file.
- Three of its constructors:

  JFileChooser( )

  JFileChooser(File *dir*)

  JFileChooser(String *dir*)
- The first one initially displays the default directory.
- If *dir* is null, the default directory is used.

Displaying a JFileChooser

- You display a **JFileChooser** using one of these methods (all throw **HeadlessException**):

  int showOpenDialog(Component *parent*)

  int showSaveDialog(Component *parent*)

  int showDialog(Component *parent*, String *name*)
- The third method lets you specify the name of the button indicating approval.

Returning from a JFileChooser

The 3 methods return one of these **JFileChooser** constants:

APPROVE_OPTION	The user selected a file.
CANCEL_OPTION	The user canceled by clicking the Cancel button or the close box.
ERROR_OPTION	An error was encountered.

Getting the Chosen File

- To get the selected file, use the **JFileChooser** method:

  File getSelectedFile( )

- You can allow directories to be selected and allow more than one file to be selected using the **JFileChooser** methods:

  void setFileSelectionMode(int *fsm*)

  void setMultiSelectionEnabled(boolean *on*)