Declarations

Class Declarations

A class declaration introduces a new reference type. It has the following general syntax:

```
<class modifiers> class <class name><formal type parameter list>
<extends clause> <implements clause>  // Class header
{ // Class body
<field declarations>
<method declarations>
<nested class declarations>
<nested interface declarations>
<nested enum declarations>
<constructor declarations>
<initializer blocks>
}
```

Class header

In the class header, the name of the class is preceded by the keyword class.

In addition, the class header can specify the following information:

- accessibility modifier
- additional class modifiers
- a formal type parameter list, if the class is generic
- any class it extends
- any interfaces it implements

Class body

- field declarations
- method declarations
- nested class, enum, and interface declarations
- constructor declarations
- static and instance initializer blocks

Static context and Non-static context

- A static context is defined by static methods, static field initializers, and static initializer blocks.

- A non-static context is defined by instance methods, constructors, non-static field initializers, and instance initializer blocks.

Static code can only refer to other static members!

JavaBeans Standard

The JavaBeans Standard allows reusable software components to be modelled in Java so that these components can be assembled to create sophisticated applications.

Naming Patterns for Properties

Example:

```
public class Light {
  // Properties:
  private int     noOfWatts;       // wattage
  private String  location;        // placement
  private boolean indicator;       // on or off
  // Setters
  public void setNoOfWatts(int noOfWatts)  { this.noOfWatts = noOfWatts; }
  public void setLocation(String location) { this.location = location; }
  public void setIndicator(boolean indicator) { this.indicator = indicator; }
  // Getters
  public int getNoOfWatts() { return noOfWatts; }
  public String  getLocation()  { return location; }
  public boolean isIndicator()  { return indicator; }
}
```

Method Declarations

```
<method modifiers> <formal type parameter list>
<return type> <method name>
                    (<formal parameter list>)
<throws clause> // Method header
{ // Method body
<local variable declarations>
<nested local class declarations>
<statements>
}
```

Method Header

- method name
- scope or accessibility modifier
- additional method modifiers
- a formal type parameter list, if the declaration is for a generic method
- the type of the return value, or void if the method does not return any value
- a formal parameter list
- checked exceptions thrown by the method are specified in a throws clause

Formal Parameter List

- comma-separated list of parameters for passing information to the method when the method is invoked by a method call
- An empty parameter list must be specified by ()
- Each parameter is a simple variable declaration consisting of its type and name:

<parameter modifier> <type> <parameter name>

The Signature of Method

The signature of a method comprises the method name and the formal parameter list only

The Method Body

- The method body is a block containing the local declarations and the statements of the method
- Local variable declarations, and nested local class declarations

Member Methods

- instance methods
- static methods

Statements

Statements in Java can be grouped into various categories.

- Variable declarations with explicit initialization of the variables are called **declaration statements** .
- Other basic forms of statements are **control flow statements** and
- **expression statements**.

Labeled statements are discussed later

Expression Statements

- An expression statement is an expression terminated by a semicolon.
- The expression is evaluated for its side effect and its value discarded.
- Only certain types of expressions have meaning as statements.

Expression Statements

- assignments
- increment and decrement operators
- method calls
- object creation expressions with the new operator

A solitary semicolon denotes the empty statement that does nothing.

Block

- A block, {}, is a compound statement which can be used to group zero or more local declarations and statements .
- Blocks can be nested, since a block is a statement that can contain other statements.
- A block can be used in any context where a simple statement is permitted.
- The compound statement which is embodied in a block, begins at the left brace, {, and ends with a matching right brace, }.
- Such a block must not be confused with an array initialization block in declaration statements

Instance Methods and the Object Reference this

- All members defined in the class, both static and non-static, are accessible in the context of an instance method.
- All instance methods are passed an implicit reference to the current object – this.
- In the body of the method, the this reference can be used like any other object reference to access members of the object.

Instance Methods and the Object Reference this

- The this reference can be used as a normal reference to reference the current object, but the reference cannot be modified—it is a final reference
- The this reference to the current object is useful in situations where a local variable hides, or shadows, a field with the same name.

Example: Using the this Reference

```
public class Light {
  // Fields:
  int     noOfWatts;      // wattage
  boolean indicator;      // on or off
  String  location;       // placement
  // Constructor
  public Light(int noOfWatts, boolean indicator, String site) {
    String location;
    this.noOfWatts = noOfWatts;   // (1) Assignment to field.
    indicator = indicator;        // (2) Assignment to parameter.
    location = site;              // (3) Assignment to local variable.
    this.superfluous();           // (4)
    superfluous();                // equivalent to call at (4)
  }
  public void superfluous() { System.out.println(this); }  // (5)
  public static void main(String[] args) {
    Light light = new Light(100, true, "loft");
    System.out.println("No. of watts: " + light.noOfWatts);
    System.out.println("Indicator: " + light.indicator);
    System.out.println("Location: "  + light.location);
  }
}
```

Method Overloading

Example: java.lang.Math contains

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

Method Overloading

```
void bake(Cake k)  { /* ... */ }        // (1)
void bake(Pizza p) { /* ... */ }        // (2)
int     halfIt(int a) { return a/2; }  // (3)
double  halfIt(int a) { return a/2.0; }// (4)
// Not OK. Same signature.
```

Constructors

```
<accessibility modifier> <class name>
(<formal parameter list>)
<throws clause>  // Constructor header
{ // Constructor body
<local variable declarations>
<nested local class declarations>
<statements>
}
```

Constructors

- Modifiers other than an accessibility modifier are not permitted in the constructor header.
- Constructors cannot return a value and, therefore, do not specify a return type, not even void, in the constructor header. But their declaration can use

the return statement that does not return a value in the constructor body.

- The constructor name must be the same as the class name.

Bad Example

```
public class Name {
  Name() {              // (1)
    System.out.println("Constructor");
  }
  void Name() {           // (2)
    System.out.println("Method");
  }
  public static void main(String[] args) {
    new Name().Name();
    // (3) Constructor call followed by method call.
  }
}
```

The Default Constructor

A default constructor is a constructor without any parameters

If a class does not specify any constructors, then an implicit default constructor is generated for the class by the compiler. The implicit default constructor is equivalent to the following implementation:

```
<class name>() { super(); }
```

Constructors

- A class can choose to provide an implementation of the default constructor.
- The explicit default constructor ensures that any object created with the object creation expression
- If a class defines any explicit constructors, it can no longer rely on the implicit default constructor to set the state of its objects.
- If such a class requires a default constructor, its implementation must be provided

Overloaded Constructors

```
class Light {
  // ...
  // Explicit Default Constructor:
  Light() {                // (1)
    noOfWatts = 50;
    indicator = true;
    location  = "X";
  }
  // Non-default Constructor:
  Light(int noOfWatts, boolean indicator, String location) { // (2)
    this.noOfWatts = noOfWatts;
    this.indicator = indicator;
    this.location  = location;
  }
  //...
}
class Greenhouse {
  // ...
  Light moreLight  = new Light(100, true, "Greenhouse");     // (3) OK.
  Light firstLight = new Light();            // (4) OK.
}
```

Declaring Typesafe Enums

The canonical form of declaring an enum type is shown below.

```
enum MachineState { BUSY, IDLE, BLOCKED }
```

Using Typesafe Enums

```
// Filename: MachineState.java
public enum MachineState { BUSY, IDLE, BLOCKED }

// Filename: Machine.java
public class Machine {
  private MachineState state;
  public void setState(MachineState state) {
      this.state = state;
  }
  public MachineState getState() {
      return this.state;
  }
}
```

Using Typesafe Enums

```
// Filename: MachineClient.java
public class MachineClient {
  public static void main(String[] args) {
    Machine machine = new Machine();
    machine.setState(MachineState.IDLE);  // (1) Passed as a value.
    // machine.setState(1);               // (2) Compile-time error!
    MachineState state = machine.getState(); // (3) Declaring a reference.
    System.out.println(
        "The machine state is: " + state  // (4) Printing the enum name.
    );
    // MachineState newState = new MachineState();
    // (5) Compile-time error!
  }
}
```

Declaring Enum Constructors and Members

```
// Filename: Meal.java
public enum Meal {
  BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);       // (1)
  // Non-default constructor                  (2)
  Meal(int hh, int mm) {
    assert (hh >= 0 && hh <= 23): "Illegal hour.";
    assert (mm >= 0 && mm <= 59): "Illegal mins.";
    this.hh = hh;
    this.mm = mm;
  }
  // Fields for the meal time:                (3)
  private int hh;
  private int mm;
  // Instance methods:                  (4)
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
}
```

Declaring Enum Constructors and Members

```
// Filename: MealAdministrator.java
public class MealAdministrator {
  public static void main(String[] args) {
    System.out.printf(                // (5)
        "Please note that no eggs will be served at %s, %02d:%02d.%n",
        Meal.BREAKFAST, Meal.BREAKFAST.getHour(), Meal.BREAKFAST.getMins()
    );
    System.out.println("Meal times are as follows:");
    Meal[] meals = Meal.values();             // (6)
    for (Meal meal : meals)                 // (7)
      System.out.printf("%s served at %02d:%02d%n",
   meal, meal.getHour(), meal.getMins()
      );
    Meal formalDinner = Meal.valueOf("DINNER");       // (8)
    System.out.printf("Formal dress is required for %s at %02d:%02d.%n",
        formalDinner, formalDinner.getHour(), formalDinner.getMins()
    );
  }
}
```

Implicit Static Methods for Enum Types

`static [] values()`

Returns an array containing the enum constants of this enum type, in the order

they are specified.

`static  valueOf(String name)`

Returns the enum constant with the specified name. An IllegalArgumentException is thrown if the specified name does not match the name of an enum constant.

The specified name is not qualified with the enum type name.

Inherited Methods from the Enum Class

protected final Object clone()

final int compareTo(E o)

final boolean equals(Object other)

protected final void finalize()

final Class<E> getDeclaringClass()

final int hashCode()

final String name()

final int ordinal()

Extending Enum Types: Constant-Specific Class Bodies

```
BREAKFAST(7,30) {                          // (1)
// Start of constant-specific class body
  public double mealPrice(Day day) {  // (2)
// Overriding abstract method
    ...
  }
  public String toString() {           // (3)
// Overriding method from the Enum
class
    ...
  }
}             // (4)
// End of constant-specific class body
```

Declaring Typesafe Enums Revisited

- An enum type can be declared as a top-level type.
- Enum types can also be nested, but only within other static members, or other top-level type declarations.

```
public class MealPrices {
  public enum Day { /* ... */ }           // Static member
  public static enum Meal { /* ... */ }  // Static member
  public static void main(String[] args) { /* ... */ }
// Static method
}
```

Declaring Typesafe Enums Revisited

An enum type cannot be declared abstract, regardless of whether each abstract method is overridden in the constant-specific class body of every enum constant.

Like a class, an enum can implement interfaces.

```
public interface ITimeInfo {
  public int getHour();
  public int getMins();
}
public enum Meal implements ITimeInfo {
  // ...
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
}
```

Arrays

A one-dimensional array variable declaration has either the following syntax:

```
<element type>[] <array name>;
```

or

```
<element type> <array name>[];
```

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

Constructing an Array

```
<array name> = new <element type> [<array size>];
```

The array declaration and construction can be combined.

```
<element type1>[] <array name> = new <element type2>[<array size>];
```

Constructing Array

```
int[] anIntArray = new int[10];
// Default element value: 0.
Pizza[] mediumPizzas = new Pizza[5];
// Default element value: null.
// Pizza class extends Object class
Object[] objArray = new Pizza[3];
// Default element value: null.
// Pizza class implements
//Eatable interface
Eatable[] eatables = new Pizza[2];
// Default element value: null.
```

Initializing an Array

```
<element type>[] <array name> =
{ <array initialize list> };

int[] anIntArray = {
    13, 49, 267, 15, 215
}
// Pizza class extends Object class
Object[] objArray = { new Pizza(),
new Pizza(), null };
```

Initializing an Array

// Array with 4 String objects:

```
String[] pets = {
"crocodiles", "elephants",
"crocophants", "elediles"};
```

// Array of 3 characters:

```
char[] charArray = {'a', 'h', a'};
```

Using an Array

<array name> [<index expression>]

Anonymous Arrays

```
new <element type>[] { <array initialize
list> }
```

```
int[] intArray = {3, 5, 2, 8, 6};
int[] intArray = new int[] {3, 5, 2, 8, 6};
```

Anonymous Arrays

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30,
31, 30, 31, 31, 30, 31, 30, 31};
// Not ok.
daysInMonth = new int[] {31, 28,
31, 30, 31, 30, 31, 31, 30, 31,
30, 31}; // ok.
```

Multidimensional Arrays

<element type>[][]...[] <array name>;

or

<element type> <array name>[][]...[];

```
int[][] mXnArray; // 2-dimensional array
int[] mXnArray[]; // 2-dimensional array
int mXnArray[][]; // 2-dimensional array
```

Multidimensional Arrays

```
int[][] mXnArray = new int[4][5];
// 4 x 5 matrix of ints


double[][] identityMatrix = {
   {1.0, 0.0, 0.0, 0.0 }, // 1. row
   {0.0, 1.0, 0.0, 0.0 }, // 2. row
   {0.0, 0.0, 1.0, 0.0 }, // 3. row
   {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 Floating-point matrix
```

Multidimensional Arrays

```
HotelRoom[][][][] rooms = new HotelRoom[10][5][][];
// Just streets and hotels.
rooms[0][0]       = new HotelRoom[3][];
// 3 floors in 1st. hotel on 1st. street.
rooms[0][0][0]    = new HotelRoom[8];
// 8 rooms on 1st. floor in this hotel.
rooms[0][0][0][0] = new HotelRoom();
// Initializes 1st. room on this floor.
```

Multidimensional Arrays

```
double[][] matrix = new double[3][];
// No. of rows.
for (int i = 0; i < matrix.length; ++i)
   matrix[i] = new double[i + 1];
// Construct a row.
```

Multidimensional Arrays

```
double[][] matrix2 = {
// Using array initializer blocks.
   {0.0},              // 1. row
   {0.0, 0.0},         // 2. row
   {0.0, 0.0, 0.0}     // 3. row
}
double[][] matrix3 = new double[][] {
// Using an anonymous array of arrays.
   {0.0},              // 1. row
   {0.0, 0.0},         // 2. row
   {0.0, 0.0, 0.0}     // 3. row
}
```

Parameter Passing

- <object reference>.<method name> (<actual parameter list>)
- <class name>.<static method name> (<actual parameter list>)
- <method name> (<actual parameter list>)

Parameter Passing

- `objRef.doIt(time, place);`
  `// Explicit object reference`
- `int i = java.lang.Math.abs(-1);`
  `// Fully qualified class name`
- `int j = Math.abs(-1);`
  `// Simple class name`
- `someMethod(ofValue);`
  `// Object or class implicitly implied`
- `someObjRef.make().make().make();`
  `// make() returns a reference value`

Parameter Passing

Data Type of the Formal Parameters	Value Passed
Primitive data types	Primitive data value
Class or enum type	Reference value
Array type	Reference value

Passing Primitive Data Values

Legal type conversions between actual parameters and formal parameters of primitive data types are summarized here:

- widening primitive conversion
- unboxing conversion, followed by an optional widening primitive conversion

Passing Primitive Values

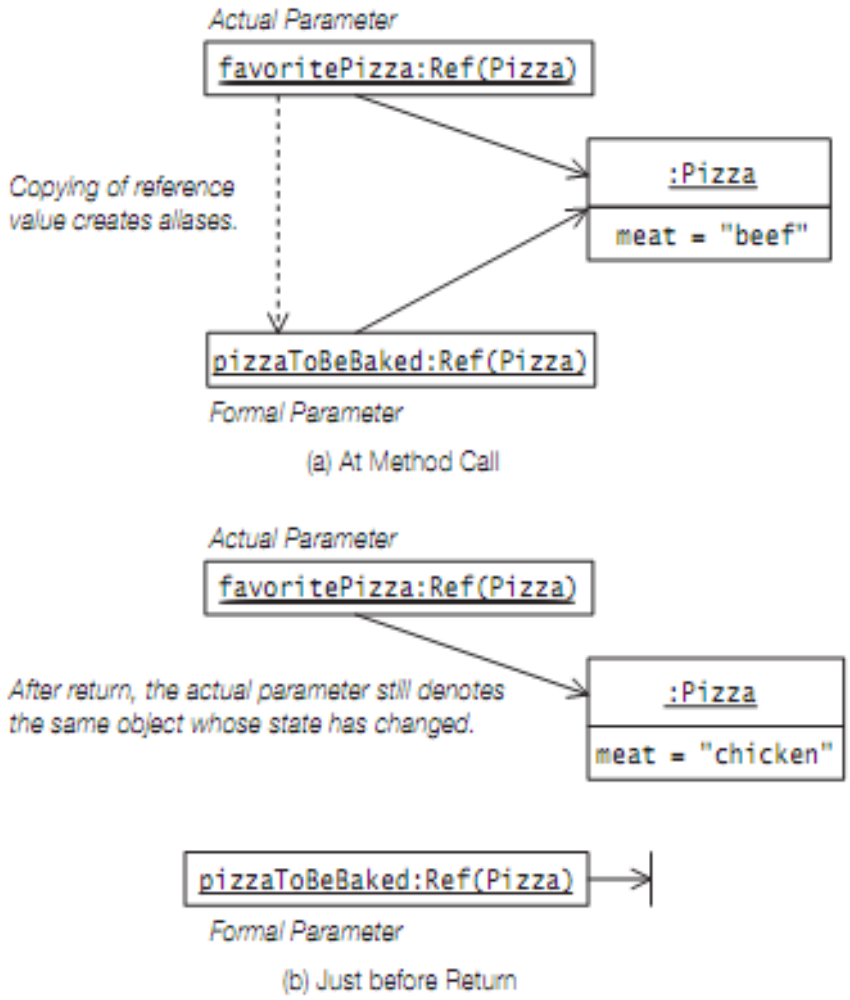
```
public class CustomerOne {
  public static void main (String[] args) {
    PizzaFactory pizzaHouse = new PizzaFactory();
    int pricePrPizza = 15;
    double totPrice = pizzaHouse.calcPrice(4, pricePrPizza);       // (1)
    System.out.println("Value of pricePrPizza: " + pricePrPizza);  // (2)
  }
}
class PizzaFactory {
  public double calcPrice(int numberOfPizzas, double pizzaPrice) {  // (3)
    pizzaPrice = pizzaPrice/2.0;                    // Changes price.  (4)
    return numberOfPizzas * pizzaPrice;
  }
}
```

Passing Reference Values

```
public class CustomerTwo {
  public static void main (String[] args) {
    Pizza favoritePizza = new Pizza();                          // (1)
    System.out.println("Meat on pizza before baking: " +
favoritePizza.meat);
    bake(favoritePizza);                                        // (2)
    System.out.println("Meat on pizza after baking: " +
favoritePizza.meat);
  }
  public static void bake(Pizza pizzaToBeBaked) {               // (3)
    pizzaToBeBaked.meat = "chicken";  // Change the meat on the pizza.
    pizzaToBeBaked = null;                                      // (4)
  }
}

class Pizza {                                                   // (5)
  String meat = "beef";
}
```

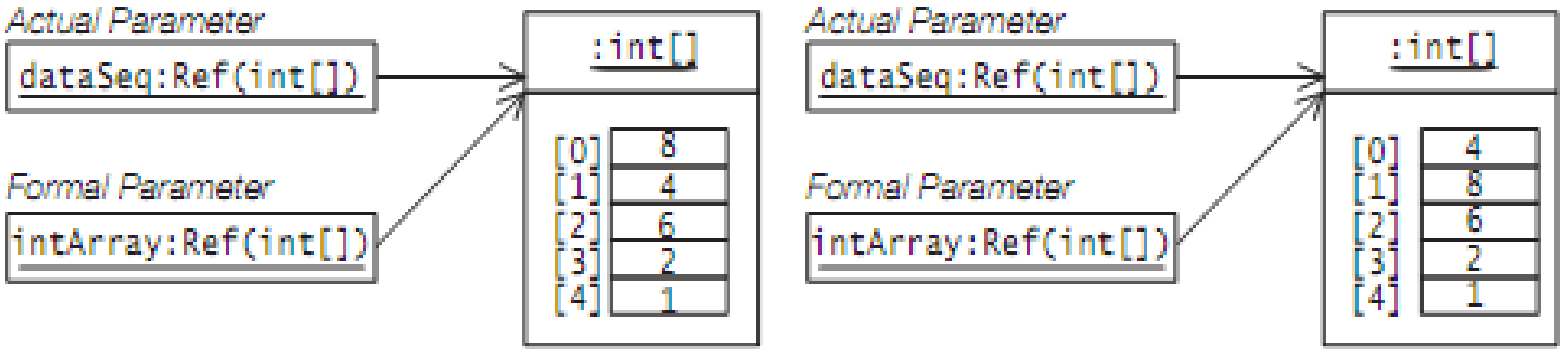
Passing Reference Values



Passing Arrays

```
public class Percolate {
  public static void main (String[] args) {
    int[] dataSeq = {6,4,8,2,1};    // Create and initialize an array.
    // Write array before percolation:
    printIntArray(dataSeq);
    // Percolate:
    for (int index = 1; index < dataSeq.length; ++index)
      if (dataSeq[index-1] > dataSeq[index])
        swap(dataSeq, index-1, index);                          // (1)
    // Write array after percolation:
    printIntArray(dataSeq);
  }
  public static void swap(int[] intArray, int i, int j) {       // (2)
    int tmp = intArray[i]; intArray[i] = intArray[j]; intArray[j] = tmp;
  }
  public static void swap(int v1, int v2) {                     // (3)
    int tmp = v1; v1 = v2; v2 = tmp;
  }
  public static void printIntArray(int[] array) {               // (4)
    for (int value : array)
      System.out.print(" " + value);
    System.out.println();
  }
}
```

Passing Arrays



(a) At first call to the swap() method

(b) Just before first return from the swap() method

Array Elements as Actual Parameters

Array Elements as Primitive Data Values

```
public class FindMinimum {
  public static void main(String[] args) {
    int[] dataSeq = {6,4,8,2,1};
    int minValue = dataSeq[0];
    for (int index = 1; index < dataSeq.length; ++index)
      minValue = minimum(minValue, dataSeq[index]);  // (1)
    System.out.println("Minimum value: " + minValue);
  }
  public static int minimum(int i, int j) {          // (2)
    return (i <= j) ? i : j;
  }
}
```

Array Elements as Actual Parameters

Array Elements as Reference Values

```
public class FindMinimumMxN {
  public static void main(String[] args) {
    int[][] matrix = { {8,4},{6,3,2},{7} };              // (1)
    int min = findMinimum(matrix[0]);                    // (2)
    for (int i = 1; i < matrix.length; ++i) {
      int minInRow = findMinimum(matrix[i]);             // (3)
      if (min > minInRow) min = minInRow;
    }
    System.out.println("Minimum value in matrix: " + min);
  }
  public static int findMinimum(int[] seq) {             // (4)
    int min = seq[0];
    for (int i = 1; i < seq.length; ++i) min = Math.min(min, seq[i]);
    return min;
  }
}
```

final Parameters

- The compiler can treat final variables as constants for code optimization purposes.
- Declaring parameters as final prevents their values from being changed inadvertently.
- Whether a formal parameter is declared as final, does not affect the caller's code.

final Parameters

```
public double calcPrice(int numberOfPizzas, final double pizzaPrice) {
    pizzaPrice = pizzaPrice/2.0;        // Not allowed.
    return numberOfPizzas * pizzaPrice;
}

public static void bake(final Pizza pizzaToBeBaked) {
  pizzaToBeBaked.meat = "chicken";        // Allowed.
  pizzaToBeBaked = null;                  // Not allowed.
}
```

Variable Arity Methods

The **last** formal parameter in a variable arity method declaration is declared as follows:

```
<type>... <formal parameter name>
```

Variable Arity Methods

```
public static void publish(int n, String... data) {
                           // (int, String[])
  System.out.println("n: " + n + ", data size: " +
                     data.length);
}
```

The varargs parameter in a variable arity method is always interpreted as having the type:

`<type>[]`

In the body of the publish() method, the varargs parameter data has the type String[], i.e., a simple array of Strings.

The main() Method

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