Preliminaries I

Agenda

Primitive Java

- Primitive types
- Operators
- Statements
- Methods

Reference types

- Strings
- Arrays
- Exception handling
- Input and output

The first program

```
 1 // First program
 2 // MW, 5/1/10
 3
 4 public class FirstProgram
 5 {
 6     public static void main( String [ ] args )
 7     {
 8         System.out.println( "Is there anybody out there?" );
 9     }
10 }
```

figure 1.1

A simple first program

Compilation: `javac FirstProgram.java`

Execution: `java FirstProgram`

Three important concepts related to languages

- **Syntax** (grammar)
- **Semantics** (meaning)
- **Pragmatics** (use)

How are these concepts related to the sentence “I am hungry”?

On the need for grammar

Alan Creak (2003)

When learning our first natural language we spend long periods hearing the language (or noise) and seeing it in the context of other people's actions, and from this we learn the structure of the language, and how it is related to what happens in the world. We might not know the rules, but we learn how the language machine works.

In programming, not only have we no experience of the world we're in; we have no experience of others operating in that world, so we are stuck with grammar.

Types

A **type** is a set of legal values

A **variable** refers to a location in memory where a value can be stored

Each variable is associated with a type. The variable type restricts the values that the variable may hold.

The types in Java are divided into two categories:

(1) Primitive types

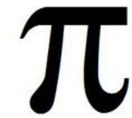
(2) Reference types

Primitive types

Primitive Type	What It Stores	Range
`byte`	8-bit integer	–128 to 127
`short`	16-bit integer	–32,768 to 32,767
`int`	32-bit integer	–2,147,483,648 to 2,147,483,647
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit floating-point	6 significant digits (10^{-46}, 10^{38})
`double`	64-bit floating-point	15 significant digits (10^{-324}, 10^{308})
`char`	Unicode character	
`boolean`	Boolean variable	`false` and `true`

figure 1.2

The eight primitive types in Java



Constants (literals)

An integer constant may be written in

decimal notation: 23
octal notation: 027
hexadecimal notation: 0x17

Floating-point constants are written as a decimal number with an optional exponent part:

3.24 3.24f 3.24d
3.24e5 3.24e-5

Character constants are written directly between single quotes:

'z' '\172' '\u007A'
'\n' '\t'
'\'' '\"' '\\'

Declaration and initialization

Any variable is declared by providing its type, its name, and optionally, its initial value.

Examples:

```
int num1;
double minimum = 4.50;
int x = 0, num2 = 2;
int num3 = 2 * num2;
```

Basic operators

```
1  public class OperatorTest
2  {
3      // Program to illustrate basic operators
4      // The output is as follows:
5      // 12 8 6
6      // 6 8 6
7      // 6 8 14
8      // 22 8 14
9      // 24 10 33
10
11     public static void main( String [ ] args )
12     {
13         int a = 12, b = 8, c = 6;
14
15         System.out.println( a + " " + b + " " + c );
16         a = c;
17         System.out.println( a + " " + b + " " + c );
18         c += b;
19         System.out.println( a + " " + b + " " + c );
20         a = b + c;
21         System.out.println( a + " " + b + " " + c );
22         a++;
23         ++b;
24         c = a++ + ++b;
25         System.out.println( a + " " + b + " " + c );
26     }
27 }
```

figure 1.3

Program that illustrates operators

`2 + 3 * 4` = ?

Arithmetic operators

Precedence and associativity

Category	**Operator**	**Associativity**
Unary	`++ -- + -`	*right-to-left*
Multiplicative	`* / %`	left-to-right
Additive	`+ -`	left-to-right
Assignment	`= += -= *= /= %=`	*right-to-left*

`++`: increment
`--`: decrement
`%` : remainder (modulus)

Type conversion

Type conversion is the conversion of values of one type to values of another type

Explicit type conversion (*casting*) is to generate a temporary entity of another type.

Example:

```
int x = 6, y = 10;
double quotient = x / y;   // Probably wrong!
```

Replace by:

```
double quotient = ((double) x) / y;
```

Widening and narrowing of types

Converting a type of a smaller range to a type of a larger range is called **widening**

Converting a type of a larger range to a type of a smaller range is called **narrowing**

Example:

```
int i = 10;
long m = 10000;
double d = Math.PI;
i = (int) m;   // narrowing (cast necessary)
m = i;         // widening
m = (long) d;  // narrowing (cast necessary)
d = m;         // widening
```

Statements

Simple statements:

Expression statements (assignment expressions, method invocation, increment/decrement expressions)

Variable declarations

`break` statements

`continue` statements

`return` statements

Compound statements:

Statement blocks (statements enclosed by a pair of braces, `{ }`)

Selection statements (`if`, `switch`)

Loop statements (`for`, `while`, `do`)

`try-catch` statements

Expression statements

Examples:

```
x = y;
x = y = 3;
x = x + 1;
x++;
y = y + x;
y += x;
move(dx, dy);
```

Declaration statements

The placement of a declaration determines its **scope** (the extent of the code in which the variable is visible, that is accessible).

Example:

```
{
    ...
    int i;       // i’s scope begins here
    i = 10;
    int j = 20; // j’s scope begins here
    i += j;
    ...
} // both i's and j’s scope end here
```

`if` statements

```
if (Condition)
    Statement
```

or

```
if (Condition)
    Statement1
else
    Statement2
```

Condition must be a boolean expression

Logical operators

figure 1.4

Result of logical operators

x	y	x && y	x || y	!x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

switch statements

```
switch (Expression) {
case CaseLabel1:
    Statement1
  .
  .
  .
case CaseLabeln:
    Statementn
default:
    Statementn+1
}
```

Expression must be an integer or string expression.
The case labels must be *constant* integer or string expressions.
The default clause is optional.

Example of `switch` statement

figure 1.5

Layout of a `switch` statement

```
 1 switch( someCharacter )
 2 {
 3   case '(':
 4   case '[':
 5   case '{':
 6     // Code to process opening symbols
 7     break;
 8 
 9   case ')':
10   case ']':
11   case '}':
12     // Code to process closing symbols
13     break;
14 
15   case '\n':
16     // Code to handle newline character
17     break;
18 
19   default:
20     // Code to handle other cases
21     break;
22 }
```

Loop statements

```
while (Condition)
    Statement
```

```
do
    Statement
while (Condition);
```

```
for (InitExpr; Condition; UpdateExpr)
    Statement
```

is (usually) equivalent to

```
InitExpr;
while (Condition) {
    Statement
    UpdateExpr;
}
```

```
for (type var : Collection)
    Statement
```

for-each loop
←

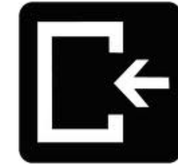
Use of the `for` statement

The standard use of the for statement is to run a code block a **specified number** of times.

```
for (int i = 0; i < n; i++)
    statement;
```

The initializer and updater expressions may be comma-separated lists of expressions:

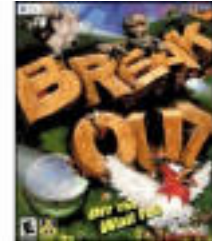
```
for (int i = 0, j = 2 * n; i < n; i++, j -= 3)
```



Typical use of the `do` statement

```
do {
   prompt user;
   read value;
} while (value is no good);
```

Use of **`break`** in loops



```
while (...) {
    ...
    if (something)
        break;
    ...
}
```

```
outerLoop:
while (...) {
    while (...) {
        if (disaster)
            break outerLoop;
    }
}
```

Use of `continue` in loops

```
for (int i = 1; i <= 100; i++) {
    if (i % 10 == 0)
        continue;
    System.out.println(i);
}
```

The conditional operator ? :

```
max_ab = a > b ? a : b;
```

is equivalent to

```
if (a > b)
    max_ab = a;
else
    max_ab = b;
```

Method declarations and calls

```
 1 public class MinTest
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         int a = 3;
 6         int b = 7;
 7
 8         System.out.println( min( a, b ) );
 9     }
10
11     // Method declaration
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

figure 1.6

Illustration of method declaration and calls

Parameter passing

In Java, all parameters are passed **by value**. In other words, modifications to parameters inside a method will be made to copies of the actual parameters and will have no effect on the actual parameters themselves.

```
void inc(int i) {
    i++;
}
```

```
int i = 0;
inc(i);
// i is still zero
```

Overloading of method names

Java allows methods to share the same name.
The name is said to be **overloaded** with multiple implementations.

The legality of overloading depends on the **signature** of the method. The signature of a method consists of its name and a list of the types of the parameters.

Note that the return type and parameter names are not part of the signature.

Signatures

Method	Signature
`String toString()`	`toString()`
`void move(int dx, int dy)`	`move(int, int)`
`void paint(Graphics g)`	`paint(Graphics)`

Two methods in the same class can be overloaded if they have different signatures.

Overloading example

```
int max(int a, int b) { ... }

int max(int a, int b, int c) { ... }
```

Overloading is allowed if the compiler can deduce which of the intended meanings should be applied based on the actual argument types or number of arguments.

Reference types

A **reference variable** is a variable that somehow stores the memory address where an object or an array is located.*

A reference variable may also hold a special value, `null`, which indicates that no object or array is being referenced.

* Arrays are actually objects too.

Objects

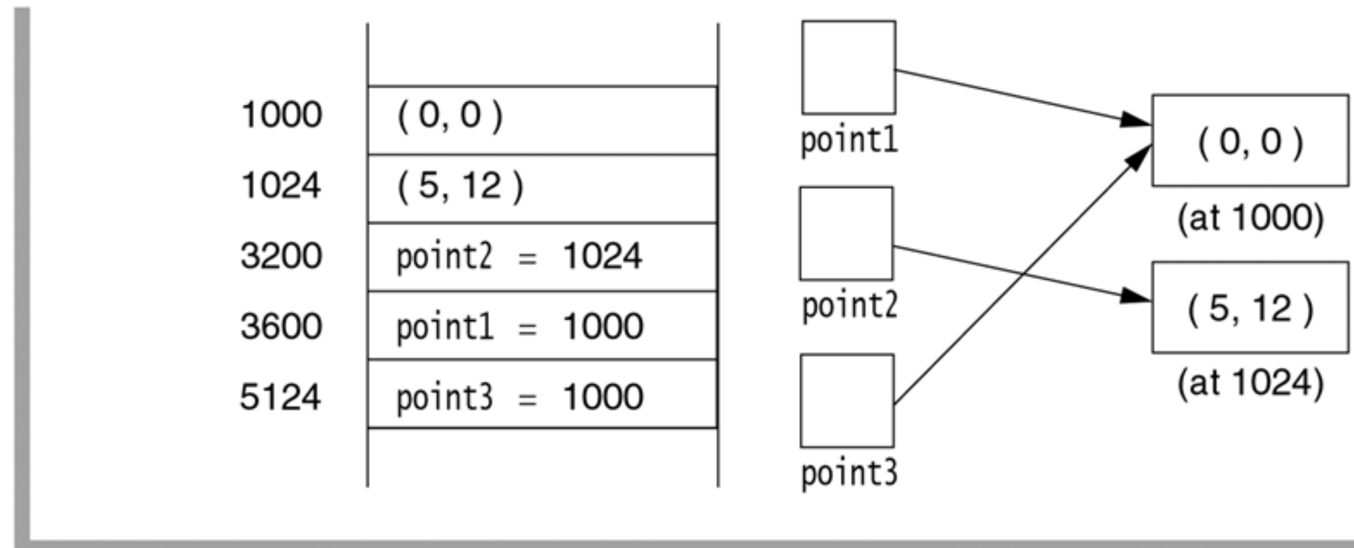
In Java, an *object* is an instance of any of the non-primitive types.

Reference variables store references to objects. The actual object is stored somewhere in memory, and the reference variable stores the object's memory address. Thus, a reference variable simply represents the name for that part of the memory.

Illustration of references

figure 2.1

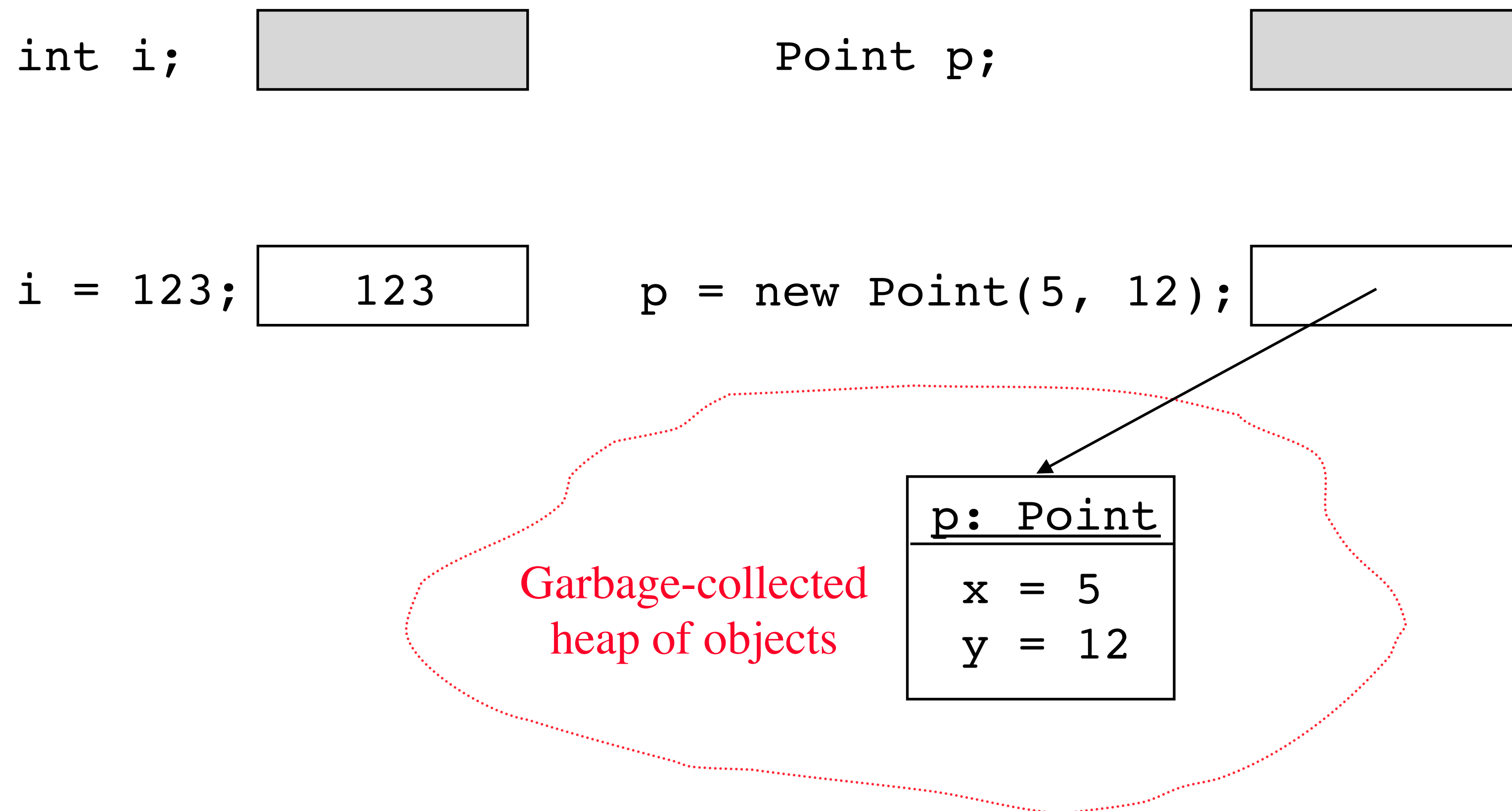
An illustration of a reference. The `Point` object stored at memory location 1000 is referenced by both `point1` and `point3`. The `Point` object stored at memory location 1024 is referenced by `point2`. The memory locations where the variables are stored are arbitrary.




```
class Point {
    int x, y;
}
```

```
Point point1, point2, point3;
```

Primitive versus reference types



The assignment operator for reference variables

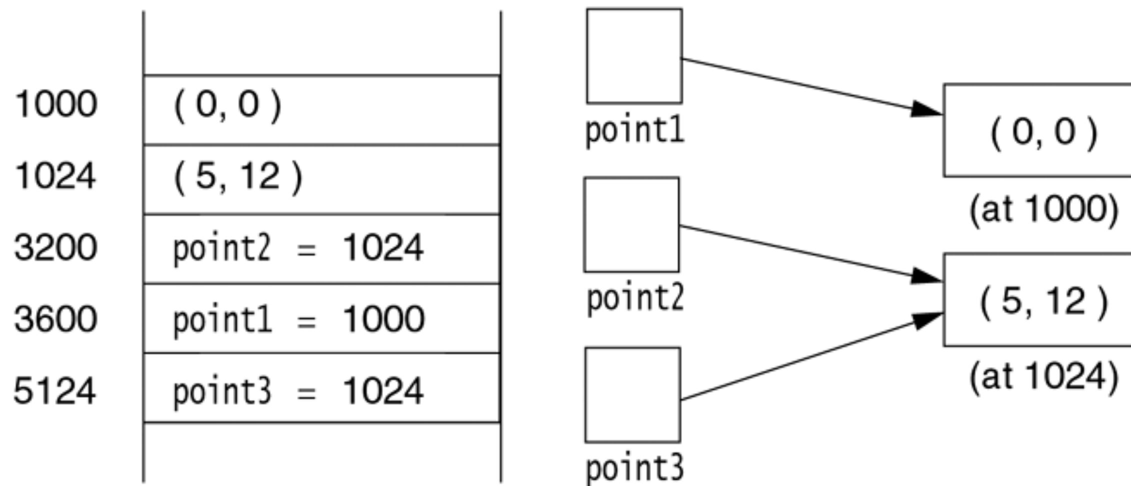


figure 2.2

The result of `point3=point2`: `point3` now references the same object as `point2`.

The dot operator

The dot operator (.) is used to select a method that is applied to an object.

```
double theArea = theCircle.area();
```

The dot operator operator can also be used to access individual components of an object, provided arrangement has been made to allow internal components to be viewable.

```
double r = theCircle.radius;
```

Declaration of reference variables

```
Button b;          // b may reference a Button object
b = new Button();  // Now b refers to an allocated object
b.setLabel("No");  // The button’s label is set to "No"
p.add(b);          // and the button is added to panel p
```

It happens that the `Button` object can be constructed with a `String` that specifies the label:

```
Button b = new Button("No");
p.add(b);    // Add it to panel p
```

Or, if the `Button` reference is not needed:

```
p.add(new Button("No"));
```

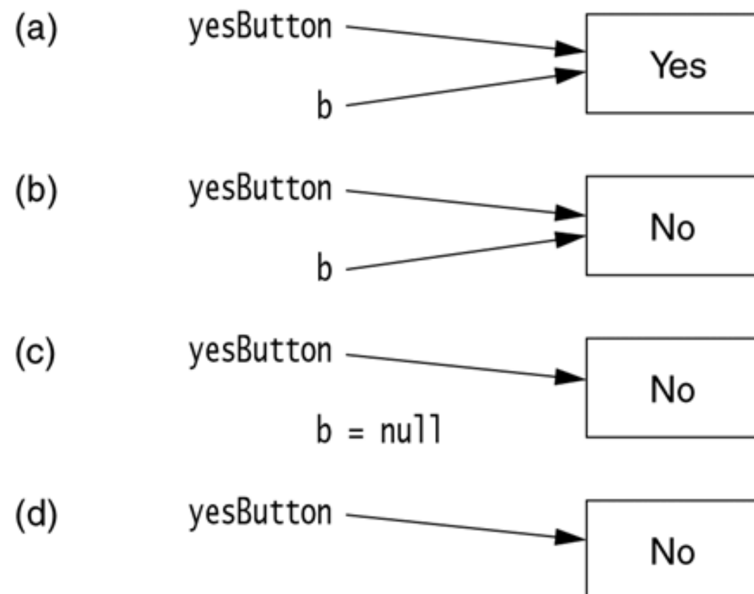
Parameter passing

```
Button yesButton = new Button("Yes");
clearButton(yesButton);
```

```
public void clearButton(Button b) {
    b.setLabel("No");
    b = null;
}
```

figure 2.3

The result of call-by-value. (a) b is a copy of yesButton; (b) after b.setLabel("No"): changes to the state of the object referenced by b are reflected in the object referenced by yesButton because these are the same object; (c) after b=null: change to the value of b does not affect the value of yesButton; (d) after the method returns, b is out of scope.



The meaning of ==

For primitive types, `a == b` is `true` if the values of `a` and `b` are identical.

For reference types, `r1 == r2` is `true` if `r1` and `r2` reference the same object.

That is, `r1 == r2` tests the identity of two *references*, not the equality of the *states* of the objects referred to by them. To test the equality (of the states) of two objects, the `equals()` method should be used, such as `r1.equals(r2)`.

Strings

A **string** is a sequence of characters

Java provides three classes to support strings:

(1) `String`: immutable (that is, constant) strings

(2) `StringBuffer`: mutable strings

(3) `StringBuilder`: mutable strings (more efficient than `StringBuffer`, but not thread-safe)

Basics of string manipulation

Declaration and initialization:

```
String empty = "";
String message = "Hello";
String repeat = message;
```

String concatenation:

```
"this" + " that" // Generates "this that"
"abc" + 5        // Generates "abc5"
5 + "abc"        // Generates "5abc"
"a" + "b" + "c"  // Generates "abc"
"a" + 1 + 2      // Generates "a12"
1 + 2 + "a"      // Generates "3a"
1 + (2 + "a")    // Generates "12a"
```

String operations

```
s.length()
s.charAt(i)
s.indexOf(c)
s.indexOf(c, i)
s1.indexOf(s2)
s1.indexOf(s2, i)
s.substring(i)
s.substring(i, j)
```

```
s.toLowerCase()
s.toUpperCase()
s.toCharArray()
s.trim()
s1.endsWith(s2)
s1.startsWith(s2)
s1.equals(s2)
s1.equalsIgnoreCase(s2)
s1.compareTo(s2)
s.intern()
s.split(regex)
```

Converting other types to strings

String concatenation provides a lazy way to convert any primitive type to a string. For instance, `"" + 45.3` returns a newly constructed `String` `"45.3"`.

All reference types provide an implementation of the method `toString` that gives a string representation of their objects.

The `int` value that is represented by a `String` can be obtained by calling the method `Integer.parseInt`. The method generates an exception if the `String` does not represent an `int`.

```
int    x = Integer.parseInt("75");
double y = Double.parseDouble("3.14");
```

Arrays

An **array** is a basic mechanism for storing a collection of *identically typed* entities.

Each entity in the array can be accessed via the array indexing operator `[]`. In Java, arrays are always indexed starting at zero. Thus an array `a` of three items stores `a[0]`, `a[1]`, and `a[2]`.

The number of items that can be stored in an array `a` can be obtained by `a.length`. Note there are no parentheses. A typical array loop would use

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

One-dimensional arrays

Declaration

Type[] *Identifier* Ex. `int[] a;`

Creation with `new`:

new *Type*[*n*] Ex.
```
int[] a = new int[10];
Point[] p = new Point[20];
```

or with an initializer list:

{ v_0, v_1, ..., v_{n-1} } Ex. `int[] a = { 7, 9, 13 };`

Simple demonstration of arrays

```
1  import java.util.Random;
2
3  public class RandomNumbers
4  {
5      // Generate random numbers (from 1-100)
6      // Print number of occurrences of each number
7
8      public static final int DIFF_NUMBERS     =     100;
9      public static final int TOTAL_NUMBERS    = 1000000;
```

figure 2.4

Simple demonstration of arrays

Output when DIFF_NUMBERS = 10:
1: 99631
2: 99883
3: 100027
4: 99932
5: 100181
6: 99946
7: 99801
8: 100528
9: 100209
10: 99862

Dynamic array expansion

Suppose we want to read a sequence of numbers and store them in an array for processing. If we have no idea how many numbers to expect, then it is difficult to make a reasonable choice of array size.

The following slide shows how to expand arrays if the initial size is too small.

Suppose we have made the declaration

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

but we find out that we really need 12 `ints` instead of 10.

```
int[] original = arr;           // 1. Save reference to arr
arr = new int[12];              // 2. Have arr reference more memory
for (int = 0; i < 10; i++)      // 3. Copy the old data over
    arr[i] = original[i];
original = null;                // 4. Unreference original array
```

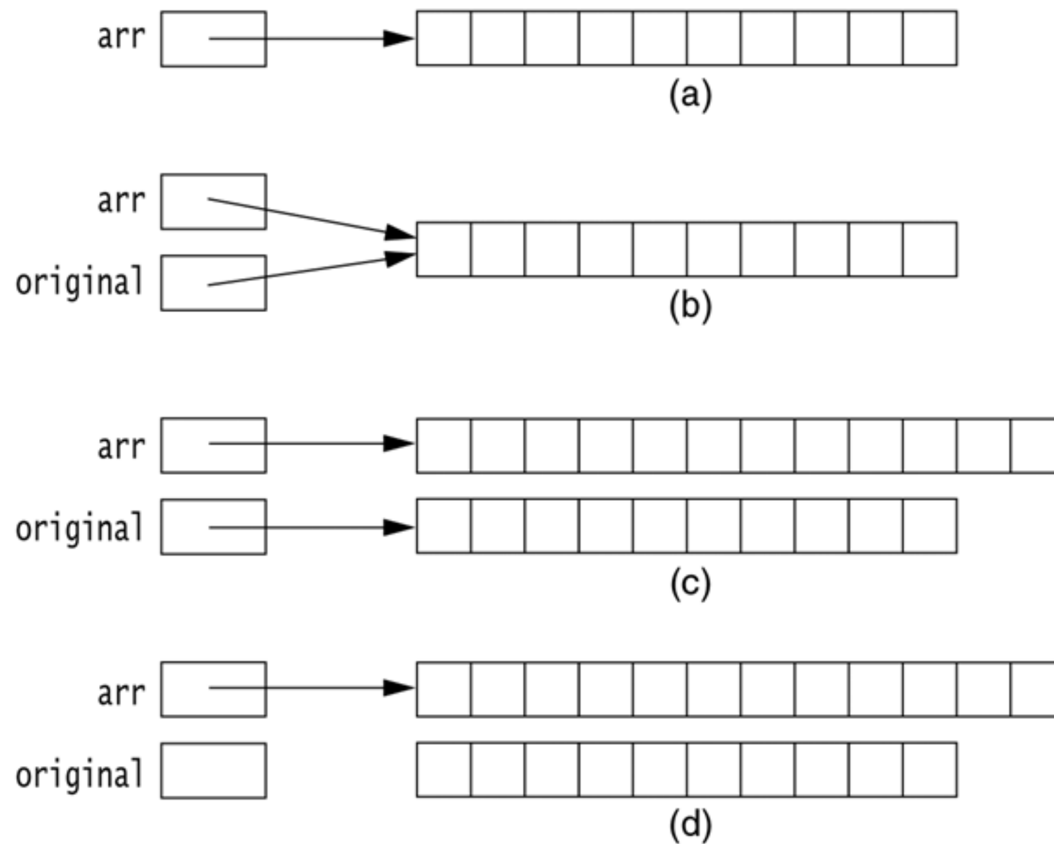


figure 2.5

Array expansion, internally: (a) At the starting point, `arr` represents 10 integers; (b) after step 1, `original` represents the same 10 integers; (c) after steps 2 and 3, `arr` represents 12 integers, the first 10 of which are copied from `original`; and (d) after step 4, the 10 integers are available for reclamation.

figure 2.6

Code to read an unlimited number of Strings and output them (part 1)

```
1  import java.util.Scanner;
2
3  public class ReadStrings
4  {
5      // Read an unlimited number of String; return a String [ ]
6      // The minimal I/O details used here are not important for
7      // this example and are discussed in Section 2.6.
8      public static String [ ] getStrings( )
9      {
10         Scanner in = new Scanner( System.in );
11         String [ ] array = new String[ 5 ];
12         int itemsRead = 0;
13
14         System.out.println( "Enter strings, one per line; " );
15         System.out.println( "Terminate with empty line: " );
16
17         while( in.hasNextLine( ) )
18         {
19             String oneLine = in.nextLine( );
20             if( oneLine.equals( "" ) )
21                 break;
22             if( itemsRead == array.length )
23                 array = resize( array, array.length * 2 );
24             array[ itemsRead++ ] = oneLine;
25         }
26
27         return resize( array, itemsRead );
28     }
```

```
38      // Resize a String[ ] array; return new array
39      public static String [ ] resize( String [ ] array,
40                                       int newSize )
41      {
42          String [ ] original = array;
43          int numToCopy = Math.min( original.length, newSize );
44
45          array = new String[ newSize ];
46          for( int i = 0; i < numToCopy; i++ )
47              array[ i ] = original[ i ];
48          return array;
49      }
50
51      public static void main( String [ ] args )
52      {
53          String [ ] array = getStrings( );
54          for( int i = 0; i < array.length; i++ )
55              System.out.println( array[ i ] );
56      }
57  }
```

figure 2.7

Code to read an unlimited number of `Strings` and output them (part 2)

`ArrayList`

The array expansion technique is so common that the Java library contains an `ArrayList` type with built-in functionality to mimic it.

The `add` method is used to add an element to the `ArrayList`. This is trivial if capacity has not been reached. If it has, the capacity is automatically expanded.

The `get` method is used to access an object at a given index.

```
 1 import java.util.Scanner;
 2 import java.util.ArrayList;
 3
 4 public class ReadStringsWithArrayList
 5 {
 6     public static void main( String [ ] args )
 7     {
 8         ArrayList<String> array = getStrings( );
 9         for( int i = 0; i < array.size( ); i++ )
10             System.out.println( array.get( i ) );
11     }
12
13     // Read an unlimited number of String; return an ArrayList
14     // The minimal I/O details used here are not important for
15     // this example and are discussed in Section 2.6.
16     public static ArrayList<String> getStrings( )
17     {
18         Scanner in = new Scanner( System.in );
19         ArrayList<String> array = new ArrayList<String>( );
20
21         System.out.println( "Enter any number of strings, one per line; " );
22         System.out.println( "Terminate with empty line: " );
23
24         while( in.hasNextLine( ) )
25         {
26             String oneLine = in.nextLine( );
27             if( oneLine.equals( "" ) )
28                 break;
29
30             array.add( oneLine );
31         }
32
33         System.out.println( "Done reading" );
34         return array;
35     }
36 }
```

```
for (String s : getStrings())
    System.out.println(s);
```

figure 2.8

Code to read an unlimited number of Strings and output them, using an ArrayList

Multidimensional arrays

(array of arrays)

Declaration

Type[]...[] *Identifier* Ex. `int[][] a`

Creation with `new`:

new *Type*[n_1][n_2]...[n_k] Ex.

```
int[][] a = new int[2][3];
Point[][] p = new Point[4][5];
```

or with an initializer list:

{ I_0, I_1, ..., I_{k-1} } Ex. `int[][] a = {{3, 1, 7}, {6, 3, 2}};`

figure 2.9

Printing a two-dimensional array

```
 1 public class MatrixDemo
 2 {
 3     public static void printMatrix( int [ ][ ] m )
 4     {
 5         for( int i = 0; i < m.length; i++ )
 6         {
 7             if( m[ i ] == null )
 8                 System.out.println( "(null)" );
 9             else
10             {
11                 for( int j = 0; j < m[i].length; j++ )
12                     System.out.print( m[ i ][ j ] + " " );
13                 System.out.println( );
14             }
15         }
16     }
17
18     public static void main( String [ ] args )
19     {
20         int [ ][ ] a = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
21         int [ ][ ] b = { { 1, 2 }, null, { 5, 6 } };
22         int [ ][ ] c = { { 1, 2 }, { 3, 4, 5 }, { 6 } };
23
24         System.out.println( "a: " ); printMatrix( a );
25         System.out.println( "b: " ); printMatrix( b );
26         System.out.println( "c: " ); printMatrix( c );
27     }
28 }
```

Command-line arguments

Command-line arguments are available by examining the parameter to `main` (the `String` array `args`).

For instance when the program is invoked

```
java Echo this that
```

`args[0]` references the string `"this"` and `args[1]` references the string `"that"`.

```
 1 public class Echo
 2 {
 3     // List the command-line arguments
 4     public static void main( String [ ] args )
 5     {
 6         for( int i = 0; i < args.length - 1; i++ )
 7             System.out.print( args[ i ] + " " );
 8         if( args.length != 0 )
 9             System.out.println( args[ args.length - 1 ] );
10         else
11             System.out.println( "No arguments to echo" );
12     }
13 }
```

figure 2.10

The *echo* command

The enhanced `for` loop

(The for-each loop)

Java 5 adds new syntax that allows you access each element in an array or ArrayList.

For instance, to print out the elements in `arr`, which has type `String[]`, you can write

```
for (String val : arr)
    System.out.println(val);
```

Exceptions

An **exception** represents an unexpected condition in a program.
The Java exception-handling mechanism facilitates recovery from unexpected conditions or failures.

The location at which an exception usually occurs is not where it can be reasonably dealt with. Java enables you to separate error-handling code from “regular” code.

Exceptions are objects that store information and are transmitted outside the normal return sequence. They are propagated back through the calling sequence until some method *catches* the exception. At that point the information stored in the object can be extracted to provide error handling.

Exception example program

```
 1 import java.util.Scanner;
 2
 3 public class DivideByTwo
 4 {
 5     public static void main( String [ ] args )
 6     {
 7         Scanner in = new Scanner( System.in );
 8         int x;
 9
10         System.out.println( "Enter an integer: " );
11         try
12         {
13             String oneLine = in.nextLine( );
14             x = Integer.parseInt( oneLine );
15             System.out.println( "Half of x is " + ( x / 2 ) );
16         }
17         catch( NumberFormatException e )
18           { System.out.println( e ); }
19     }
20 }
```

figure 2.11

Simple program to illustrate exceptions

Advantage of using exceptions

```
void processFile() {
    openTheFile();
    while (fileHasMoreLines) {
        readNextLineFromTheFile();
        printTheLine();
    }
    closeTheFile();
}
```

1. What will happen if the file does not exist?
2. What will happen if the file cannot be opened?
3. What will happen if reading a line fails?
4. What will happen if the file cannot be closed?

Error handling without exceptions

```
int processFile() {
    int errorCode = 0;
    int openFileErrorCode = openTheFile();
    if (openFileErrorCode == 0 {
        while (fileHasMoreLines) {
            int readLineErrorCode = readNextLineFromTheFile();
            if (readLineErrorCode == 0)
                printTheLine();
            else {
                errorCode = -4;
                break;
            }
        }
        int closeFileErrorCode = CloseTheFile();
        if (closeFileErrorCode != 0 && errorCode == 0)
            errorCode = -3;
    } else if (openFileErrorCode == -1)
         errorCode = -1;
    else if (openFileErrorCode == -2)
        errorCode = -2;
    return errorCode;
}
```

Error handling with exceptions

```
void processFile() {
    try {
        openTheFile();
        while (fileHasMoreLines) {
            readNextLineFromTheFile();
            printTheLine();
        }
    }
    catch (FileNotFoundException e) {
        doSomething();
    }
    catch (IOException e) {
        doSomethingElse();
    }
    finally {
        closeTheFile();
    }
}
```

Exceptions don’t save us the effort in finding and processing errors but give us a more elegant, short, clear and efficient way to do it.

Standard runtime exceptions

A runtime exception need not be handled

figure 2.12

Common standard runtime exceptions

Standard Runtime Exception	Meaning
`ArithmeticException`	Overflow or integer division by zero.
`NumberFormatException`	Illegal conversion of `String` to numeric type.
`IndexOutOfBoundsException`	Illegal index into an array or `String`.
`NegativeArraySizeException`	Attempt to create a negative-length array.
`NullPointerException`	Illegal attempt to use a null reference.
`SecurityException`	Run-time security violation.

Standard checked exceptions

A *checked* exception must be caught or explicitly propagated back to the calling method (using the `throws` clause)

Standard Checked Exception	Meaning
`java.io.EOFException`	End-of-file before completion of input.
`java.io.FileNotFoundException`	File not found to open.
`java.io.IOException`	Includes most I/O exceptions.
`InterruptedException`	Thrown by the `Thread.sleep` method.

figure 2.13

Common standard checked exceptions

Illustration of the `throws` clause

figure 2.14

Illustration of the `throws` clause

```
 1 import java.io.IOException;
 2
 3 public class ThrowDemo
 4 {
 5     public static void processFile( String toFile )
 6                                           throws IOException
 7     {
 8         // Omitted implementation propagates all
 9         // thrown IOExceptions back to the caller
10     }
11
12     public static void main( String [ ] args )
13     {
14         for( String fileName : args )
15         {
16             try
17               { processFile( fileName ); }
18             catch( IOException e )
19               { System.err.println( e ); }
20         }
21     }
22 }
```

The `Scanner` class

```
package java.util;

public class Scanner {
    public Scanner(File source);
    public Scanner(String source);

    public boolean hasNext();
    public boolean hasNextLine();
    public boolean hasNexttype();

    public String next();
    public String nextLine();
    public type nexttype();

    public void useDelimiter(String pattern);
}
```

Read two integers and output maximum using **`Scanner`** and no exceptions

```
import java.util.Scanner;

class MaxTestA {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter 2 ints: ");
        if (in.hasNextInt()) {
            int x = in.nextInt();
            if (in.hasNextInt()) {
                int y = in.nextInt();
                System.out.println("Max: " + Math.max(x, y));
                return;
            }
        }
        System.err.println("Error: need two ints");
    }
}
```

Read two integers and output maximum using `Scanner` and exceptions

```
import java.util.Scanner;

class MaxTestA {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter 2 ints: ");
        try {
            int x = in.nextInt();
            int y = in.nextInt();
            System.out.println("Max: " + Math.max(x, y));
        } catch (NoSuchElementException e) {
            System.err.println("Error: need two ints");
        }
    }
}
```

Read exactly two integers from the same line and output maximum using two `Scanners`

```
import java.util.Scanner;

class MaxTestA {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter 2 ints: ");
        try {
            String oneLine = in.nextLine();
            Scanner str = new Scanner(oneLine);
            int x = str.nextInt();
            int y = str.nextInt();
            System.out.println("Max: " + Math.max(x, y));
        } catch (NoSuchElementException e) {
            System.err.println("Error: need two ints");
        }
    }
}
```

Program to list the contents of text files

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class ListFiles {
    public static void main(String[] args) {
        if (args.length == 0)
            System.out.println("No files specified");
        for (String fileName : args)
            listFile(fileName);
    }

    public static void listFile(String fileName) {
        Scanner fileIn = null;
        System.out.println("FILE: " + fileName);
        try {
            fileIn = new Scanner(new File(fileName));
            while (fileIn.hasNextLine())
                System.out.println(fileIn.nextLine());
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            if (fileIn != null)
                fileIn.close( );
        }
     }
}
```

Program to double-space text files

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;

public class DoubleSpace {
    public static void main(String[] args) {
        for (String fileName : args)
            doubleSpace(fileName);
    }

    public static void doubleSpace(String fileName) {
        PrintWriter  fileOut = null;
        Scanner      fileIn = null;
        try {
            fileIn  = new Scanner(new File(fileName));
            fileOut = new PrintWriter(new File(fileName + ".ds"));
            while (fileIn.hasNextLine()
                fileOut.println(fileIn.nextLine( ) + "\n");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fileOut != null) fileOut.close();
            if (fileIn != null) fileIn.close();
        }
    }
}
```

<table>
<tr><th>Category</th><th>Examples</th><th>Associativity</th></tr>
<tr><td>Operations on References</td><td>. []</td><td>Left to right</td></tr>
<tr><td>Unary</td><td>++ -- ! - (type)</td><td>Right to left</td></tr>
<tr><td>Multiplicative</td><td>* / %</td><td>Left to right</td></tr>
<tr><td>Additive</td><td>+ -</td><td>Left to right</td></tr>
<tr><td>Shift (bitwise)</td><td><< >></td><td>Left to right</td></tr>
<tr><td>Relational</td><td>< <= > >= instanceof</td><td>Left to right</td></tr>
<tr><td>Equality</td><td>== !=</td><td>Left to right</td></tr>
<tr><td>Boolean (or bitwise) AND</td><td>&</td><td>Left to right</td></tr>
<tr><td>Boolean (or bitwise) XOR</td><td>^</td><td>Left to right</td></tr>
<tr><td>Boolean (or bitwise) OR</td><td>|</td><td>Left to right</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>Left to right</td></tr>
<tr><td>Logical OR</td><td>||</td><td>Left to right</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Right to left</td></tr>
<tr><td>Assignment</td><td>= *= /= %= += -=</td><td>Right to left</td></tr>
</table>

figure A.1

Java operators listed from highest to lowest precedence