EE3206 Java Programming and Applications

Lecturer: Dr. Derek Pao
Office: room G6514
Email: d.pao@cityu.edu.hk

References

1. *Cay Horstmann*, Big Java, latest edition, Wiley
2. *K. Sharan,* Beginning Java 8 Language Features, Lambda Expressions, Inner Classes, Threads, I/O, Collections and Streams, Apress, 2015
3. Java on-line documentation
   https://docs.oracle.com/javase/11/docs/api/
   http://docs.oracle.com/javase/tutorial/

Tentative Syllabus

- Java language
  - Object variables, primitive data types and wrapper classes
  - Mutable and immutable objects
  - Class design, interface and inheritance
  - Generic classes and methods. Collections.
  - Basic text I/O, exceptions and event handling
  - Basic graphical user interface
- Functional programming
- Stream API
- Multi-thread program design
- Introduction to network programming using TCP Socket (if time allows)

To write effective program

- Ability to think in abstract terms.
- Ability to see patterns and draw inferences from observations.
- Ability to organize and express ideas in writing.

General remarks on program design

- Explicit (meanings of statements are clear and easy to understand) is better than implicit (with hidden assumptions, or meanings not apparent).
- Simple is better than complex.
- Readability counts.
- If the implementation is hard to explain, it is a bad idea (even if the program is correct).
- If the implementation is easy to explain, it may be a good idea.
- Complicated program for solving a simple problem is generally not acceptable.
- Ad hoc and unstructured program that works on the small scale is often a problem in large systems.

Java is a comprehensive software development platform.

This course only serves as an introduction to software design using Java.

Self-learning is essential in this course.

I will introduce the fundamental aspects of Java programming so that students will be able to learn more advanced concepts in your future career as software engineer.

This course is not intended to be a first course in programming.

Students are expected to have programming experiences in C/C++, and basic knowledge in data structures and algorithms.

Object-Oriented Programming paradigm

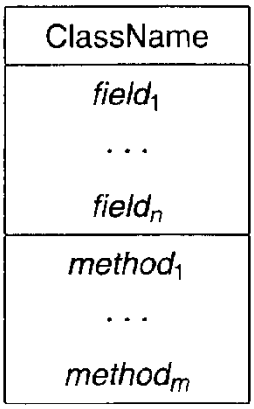
- Data encapsulation and information hiding
- Decomposition of complex system is based on the structure of objects, classes and the relationship among them
- Software reliability and reusability

Classes and Objects

	Interpretation in the real world	Representation in the model
Object	An *object* represents anything in the real world that can be distinctly identified.	An *object* is an instance of its *class*. It has a unique identity, a state, and behaviour.
Class	A *class* represents a set of objects with similar characteristics and behaviour. These objects are called the *instances* of the class.	A *class* characterizes the structure of states and behaviour that are shared by all its *instances*.

- The state of an object is composed of a set of *fields* (or *attributes*) and their current values.
- The behaviour of an object is defined by a set of *methods* (functions in terminology of C/C++), which may access or manipulate the state.

Syntax of a class definition



The top compartment shows the class name.

The middle compartment contains a list of the fields of the class.

The bottom compartment contains a list of the methods of the class.

Methods can be divided into 3 categories:

- constructor – create an object instance, assign values to instance variables
- accessor (or *getter*) – retrieve state value(s) of the object
- mutator (or *setter*) – modify (and/or retrieve) state value(s) of the object

Remarks:

- a method must be defined inside a class
- programmer needs not define destructor in a class, the JVM (Java virtual machine) reclaims memory space of unreferenced objects by a process called *garbage collection*
- static methods are utility functions, they do not operate on object instance

Example

```
public class BankAccount
{
   private double balance;
   //Instance variable - each object has its own copy.

   //The variable is declared private, hence, the users
   //of the class can only access the variable via
   //the defined public accessor/mutator methods.

   public BankAccount()  // default constructor
   {  balance = 0;
   }

   // a second constructor
   public BankAccount(double initialBalance)
   {  balance = initialBalance;
   }

   public void deposit(double amount)  // mutator
   {  balance = balance + amount;
   }

   public void withdraw(double amount) // mutator
   {  balance = balance - amount;
   }

   public double getBalance()  // accessor
   {  return balance;
   }
}
```

Sample Java application program

```
public class BankAccountTest
{
   // Java application must have a static main() method
   public static void main(String[] args)
   {
      // constructors are invoked by the new operator
      BankAccount myAccount = new BankAccount();
      BankAccount momsSavings = new BankAccount(5000.0);

      double transferAmount = 200.0;

      // transfer fund from momsSavings to myAccount
      monsSavings.withdraw(transferAmount);

      myAccount.deposit(transferAmount);

      System.out.println("balance of momsSavings = " +
                         momsSavings.getBalance());

      System.out.println("balance of myAccount = " +
                         myAccount.getBalance());
   }
}
```

Some basic syntax:

```
primitiveType varName; // scalar variable

ClassName objVarName; // declare an object variable
                      // (object reference)

objVarName = new ClassName(); // create an object instance

objVarName.methodName();// invoke an instance method
                        // on an object

ClassName.methodName();  // invoke a static method
```

Initializer block

Initializer block contains the code that is always executed whenever an instance of the class is created.

It is used to declare/initialize the common part of various constructors of a class.

```
public class BankAccount
{
   private double balance;  // instance variable

   // Initializer block
   {  balance = 0;
   }

   public BankAccount()  // default constructor
   {  // empty method
   }

   // a second constructor
   public BankAccount(double initialBalance)
   {  balance = initialBalance;
   }

   public void deposit(double amount)  // mutator
   {  balance = balance + amount;
   }

   public void withdraw(double amount)  // mutator
   {  balance = balance - amount;
   }

   public double getBalance()  // accessor
   {  return balance;
   }
}
```

Conventions of Java:

- Read the Java Coding Guidelines.
- A Java source file should have the file extension `.java`.
- A compiled source file will have `.class` extension. The binary code is called Java byte-code.
- Each source file should contain only one **`public class`**.
- The file name must be the same as the name of the `public class`.
- A source file may contain other classes that are not declared `public`.
- A Java application is a collection of classes (multiple source files). Only one class in an application contains a **`static main()`** method.
- The `main()` method represents the program entry point.
- Java byte-code is executed by the Java Virtual Machine (JVM). In the current (desktop) version of the Java Runtime Environment (JRE), Java byte-code is compiled to machine codes using the just-in-time compilation approach.
- Computation efficiency of Java program is comparable to C/C++ program.

- In the above example, the application contains two source files, `BankAccountTest.java` and `BankAccount.java`
- When `BankAccountTest.java` is compiled, the compiler sees the `BankAccount` class is being used, it will look for a file named `BankAccount.class`. If the compiler does not find that file, it automatically searches for `BankAccount.java` and compiles it.
- If the timestamp of `BankAccount.java` is newer than that of the existing `BankAccount.class`, the compiler will automatically recompile the file.

Primitive Data Types in Java

Type	**Description**	**Size**
int	The (signed) integer type, with range -2,147,483,648 to 2,147,483,647	4 bytes
byte	The type describing a single byte, with range -128 to 127	1 byte
short	The short integer type, with range -32768 to 32767	2 bytes
long	The long integer type, with range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes
double	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
float	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
char	The character type, representing code units in the Unicode encoding scheme	2 bytes
boolean	The type with the two truth values false and true	1 byte (JVM dependent)

Remarks:
Unlike C/C++, Java does not have explicit pointer type.

Java does not have unsigned integer data type.

Starting from Java 8, the class `Integer` (and class `Long`) offers support for unsigned integer arithmetic operations, e.g. `compareUnsigned(int, int)` and `divideUnsigned(int, int)`.

Scalar variables and object references

Consider the following codes

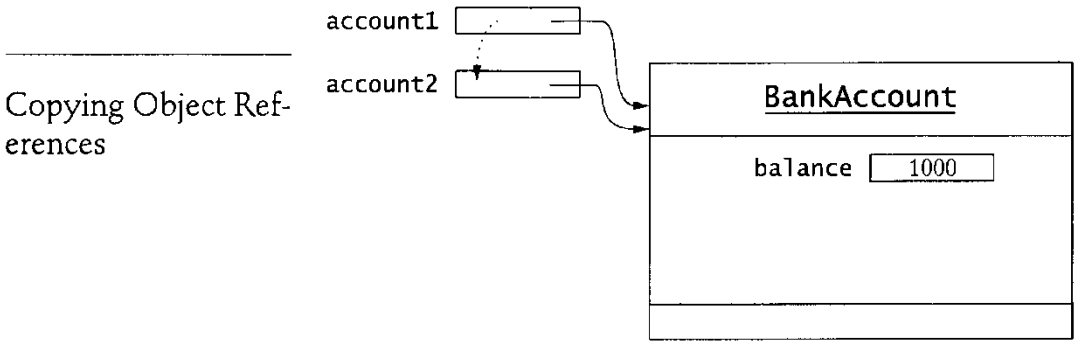
```
double balance1 = 1000;
double balance2 = balance1;  // copy data value
balance2 = balance2 + 500;
```

Now `balance1` is 1000 and `balance2` is 1500.

Consider the seemingly analogous codes with `BankAccount` objects

```
BankAccount account1 = new BankAccount(1000);
BankAccount account2 = account1; // copy object reference
account2.deposit(500);
```

The balance of both `account1` and `account2` are equal to 1500.
In fact, both `account1` and `account2` refer to the same object.



Copying Object References

Object variables do not hold values, they hold *references* to objects.

An object variable may refer to *no object*. This special reference to no object is called **`null`**.

Wrapper classes

- There is a wrapper class corresponding to each of the primitive data type.
- For example, `class Integer` is the wrapper class of `int`
- Conversion between the primitive type and the corresponding wrapper class is automatic (taken care of by the compiler).
- This process is called *auto-boxing*.

Examples

```
String num = "123";

int i = Integer.parseInt(num); // in binary: 0111 1011

Integer k = i; // same as k = new Integer(i)

int j = k;     // same as j = k.intValue()

int t = k + 1; // k.intValue() + 1

int m = null;  // Syntax error: incompatible type
// m = NULL (allowed in C/C++, equivalent to m = 0)

Integer q = null;  // OK
```

Java is a strongly typed language.

An assignment operation checks for data type compatibility.

```
double x = 2.0;

int a = x;  // syntax error, potential loss of precision
            // and possibility of overflow

int b = (int) x;  // explicit typecast required


float f = 2.5; // syntax error,
               // the constant 2.5 is represented by
               // a double (by default)

float t = (float) 2.5; // OK

float y = 2.5f;        // suffix f denotes the number is
                       // represented by a float
```

```
int flag = 1;

if (flag)   // syntax error in Java (valid in C/C++)
   // action


boolean predicate;
predicate = 0;  // syntax error in Java (valid in C/C++)

predicate = false; // correct syntax in Java
```

Enumerated type `enum` (Introduced in Java 5)

An `enum` type is a special data type that allows the programmer to represent a fixed set of constants (symbols).

```
enum Day
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY
};
```

Each constant in the enumerated list is mapped to an integer value, called the ordinal number (start from 0).

Example codes

```
Day today = Day.MONDAY;

System.out.println("Today is " + today);
// output: Today is MONDAY

System.out.println("Ordinal number of TUESDAY = " +
                   Day.TUESDAY.ordinal());
// output: Ordinal number of TUESDAY = 2


if (today == 1) // syntax error in Java (valid in C/C++)
   System.out.println("Today is MONDAY");

if (today.ordinal() == 1) // OK
   System.out.println("Today is MONDAY");

if (today == Day.MONDAY)  // OK
   System.out.println("Today is MONDAY");
```

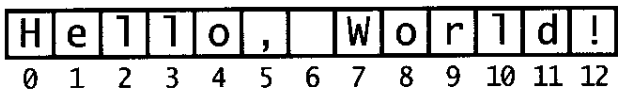
String

In Java, a character string variable is an object instance of the class `String` (defined in the package `java.lang.String`)

The `class String` has the following methods

```
int length()
// return the length of the string
String substring(int start, int pastEnd)
// return the substring
String toLowerCase() // convert to lowercase, a new String
                     // is created
String toUpperCase() // convert to uppercase
```

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 4);
   // sub is  "Hell"
```



Characters in a `String` are numbered from left to right starting from 0. Each character is a 16-bit Unicode.

You can access individual characters of a `String` using the method `charAt()`.

```
char ch = greeting.charAt(1); // value of ch is 'e'
```

No `String` method modifies the string object on which they operate. Hence, strings are *immutable objects* (value of the object instance cannot be changed after creation).

Concatenation

The + operator concatenates two strings.

```
String fname = "Harry";
String lname = "Hacker";
String name = fname + " " + lname; // name = "Harry Hacker"
```

Concatenate number to string

```
int age = 19;
String name = "Harry";
String nameAge = name + age;  // "Harry19"
```

The compiler will automatically invoke the `toString()` method to convert the integer to a string.

```
String nameAge = name + Integer.toString(age);
```

The `split()` method is a very useful method for dividing a string object into its constituent components.

```
String line = "Harry Hacker 123";
String[] token = line.split(" ");
// split the string using the substring with a space char
// " " as delimiter

// another option: line.split("\\s");
// use the white space char subclass \s as delimiter

// token[0] = "Harry";
// token[1] = "Hacker";
// token[2] = "123";
```

String tokenizer

The `StringTokenizer` class allows an application to break a string into tokens, and process the tokens one by one.

Methods available in class `StringTokenizer`

- `int countTokens()`
- `boolean hasMoreTokens()`
- `String nextToken()`
- `String nextToken(String delim)`

Example

```
String str = "this is a string";
StringTokenizer st = new StringTokenizer(str);

while (st.hasMoreToken())
{
   System.out.println(st.nextToken());
}

/* prints the following output:
     this
     is
     a
     string
*/
```

Some commonly used methods in class `String`

char	**charAt(int index)** Returns the char value at the specified index.
int	**compareTo(String anotherString)** Compares two strings lexicographically.
int	**compareToIgnoreCase(String str)** Compares two strings lexicographically, ignoring case differences.
boolean	**contains(CharSequence s)** Returns true if and only if this string contains the specified sequence of char values.
boolean	**equals(Object anObject)** Compares this string to the specified object.
boolean	**equalsIgnoreCase(String anotherString)** Compares this String to another String, ignoring case considerations.
int	**indexOf(int ch)** Returns the index within this string of the first occurrence of the specified character.
int	**indexOf(int ch, int fromIndex)** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf(String str)** Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf(String str, int fromIndex)** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
boolean	**isEmpty()** Returns true if, and only if, `length()` is 0.
int	**lastIndexOf(int ch)** Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf(int ch, int fromIndex)** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf(String str)** Returns the index within this string of the last occurrence of the specified substring.

int	**lastIndexOf(String str, int fromIndex)** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length()** Returns the length of this string.
boolean	**matches(String regex)** Tells whether or not this string matches the given regular expression.
String[]	**split(String regex)** Splits this string around matches of the given regular expression.
String[]	**split(String regex, int limit)** Splits this string around matches of the given regular expression.
String	**substring(int beginIndex)** Returns a string that is a substring of this string.
String	**substring(int beginIndex, int endIndex)** Returns a string that is a substring of this string.
char[]	**toCharArray()** Converts this string to a new character array.
String	**toLowerCase()** Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase()** Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim()** Returns a string whose value is this string, with any leading and trailing whitespace removed.
static String	**valueOf(boolean b)** Returns the string representation of the boolean argument.
static String	**valueOf(char c)** Returns the string representation of the char argument.
static String	**valueOf(char[] data)** Returns the string representation of the char array argument.
static String	**valueOf(char[] data, int offset, int count)** Returns the string representation of a specific subarray of the char array argument.

static String	**valueOf(double d)** Returns the string representation of the double argument.
static String	**valueOf(float f)** Returns the string representation of the float argument.
static String	**valueOf(int i)** Returns the string representation of the int argument.
static String	**valueOf(long l)** Returns the string representation of the long argument.
static String	**valueOf(Object obj)** Returns the string representation of the Object argument.

Remark:
If you need to work with intermediate string where the contents of the string are modified frequently, it is butter to use `StringBuilder`.

The program will be a lot more efficient.

Primitive data types for the processing of numerical data in Java

`long` : 64-bit integer, largest representable value is $2^{63} - 1$

`double` : 64-bit floating point, limited range and precision (e.g. up to 17 significant digits).

What if in an application you need to process integers that are larger than 2^{63}, or you need to process floating point numbers with precision more than 17 significant digits?

For example, we want to compute the value of π with high precision:

$\pi = 3.14159265358979323846264338327950288\ldots.$

Java classes to support the processing of numerical values beyond the range / precision supported by `long` and `double`.

`class BigInteger` : support arbitrary large integer

`class BigDecimal` : support arbitrary large real number with arbitrary precision.

Method invocation and passing parameters in method calls

```
public class Employee
{  // instance variables
   private String name;    //object reference
   private Date hireDate;  //object reference
   private double salary;  //scalar variable

   public Employee(String n, double s)  // constructor
   {  name = n;
      salary = s;
      hireDate = new Date();
       //create a Date object representing current time
   }

   public Date getHireDate()  // accessor method
   {  return hireDate;
   }

   public String getName()   // accessor method
   {  return name;
   }

   public double getSalary() // accessor method
   {  return salary;
   }

   public void payRaise(double percent) // mutator method
   {  salary *= (1 + percent/100.0);
   }

}
```

code statements in other classes that use the `Employee` class:

```
//instantiate object instances
Employee bill = new Employee("William", 10000.0);
Employee mary = new Employee("Mary", 8000.0);

double billSalary = bill.getSalary();
billSalary *= 1.1;  // this statement has no effect on
                    // the state value of bill

bill.payRaise(10.0); //increase salary of bill by 10%
// bill - implicit parameter (the object reference)
// 10.0 - explicit parameter (parameter inside the bracket)
// the method payRaise is applied to the object bill


String eName = bill.getName();

eName = eName.ToUpperCase();
//ToUpperCase() returns a new instance of String.
//This statement will not affect the state of bill
//At this point, eName and bill.name refer to two
//different String instances.


Date d = bill.getHireDate(); //retrieve hire date of bill
// accessor method does not modify the object's state
// however, if the method exposes a reference to a mutable
// item, then it risks to break data encapsulation.


d.setTime(d.getTime() - (long)(7*24*3600*1000));
//d and bill.hireDate both refer to the same Date object
//set the hire date of bill to 1 week earlier
```

Mutable Objects

- State value of the object can be changed after creation
- Instances of the `Employee` class are mutable

Immutable Objects

- State value of the object cannot be changed after creation.
- Objects of a class are immutable if the class definition does not contain any mutator method.
- Strings are immutable objects. Same for objects of wrapper classes.
- Remark: Uses of immutable objects help to enhance data integrity, but it introduces many "unreferenced" or garbage items and increases workload of the memory management module of the JVM.

How should we modify the `getHireDate()` method in order not to break data encapsulation?

```
public Date getHireDate()
{  // return a copy of the data instead of the reference
   // to the mutable object
   return (Date)hireDate.clone();
}
```

Remark:
Classes `LocalDate`, `LocalTime` and `LocalDateTime` are introduced in Java 8. Objects of these classes are immutable.

An alternative approach is to change the data type of `hireDate` to `LocalDate`.

Static methods

```
//class Math is defined to be final,
//it cannot be further subclassed or extended,
//i.e. the methods in the Math class cannot be altered by
//application programmers

public final class Math
{
    static double abs(double a)
    {
        ...
    }

    static double exp(double a)
    {
        ...
    }

    ...


}
```

Invocation of static methods

```
//static method does not operate on an object instance
//the implicit parameter is replaced by the class name
double d = Math.exp(2.5);  //d = e^2.5
```

Remark:

- static method cannot access instance variables defined in the class definition
- static method can access static variables defined in the class definition

Methods with variable number of arguments

In JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments.

This feature is called *varargs* (variable-length arguments).

Example:

```
class Test1
{
   // varargs is specified by three periods (...)
   // Within the method, varargs is treated as an array.
   static double average(double... x)
   {
      // Only 1 varargs is allowed in a method, and
      // it must be the last parameter.

      if (x.length == 0)
         return 0;

      double sum = 0;
      for (int i = 0; i < x.length; i++)
         sum += x[i];

      /* alternative syntax using for-each loop
      for (double d : x) // for each element d of type
         sum += d;       // double in collection x
      */

      return sum / x.length;
   }

   public static void main(String args[])
   {
      double a1 = average();              // a1 = 0

      double a2 = average(2.4, 3.0);      // a2 = 2.7

      double a3 = average(1, 2, 3, 4);    // a3 = 2.5
   }
}
```

Array

- Array is a sequence of values of the same type.
- An array is treated as an object in Java, and it has an attribute `length`.
- Use the **`new`** operator to create an array.
- An array has fixed length once created.
- Index values range from 0 to `length - 1`.

Table 1 Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int NUMBERS_LENGTH = 10;` `int[] numbers = new int[NUMBERS_LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int valuesLength = in.nextInt();` `double[] values = new double[valuesLength];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] names = new String[3];`	An array of three string references, all initially `null`.
`String[] friends = { "Emily", "Bob", "Cindy" };`	Another array of three strings.
`double[] values = new int[10]`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

Example:

What elements does the data array contain after the following statements?

```
double[] values = new double[10];
for (int i = 0; i < values.length; i++)
   values[i] = i * i;
```

Answer: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, but not 100

Make parallel arrays into arrays of objects

```
//Don't do this
int[] accountNumbers;
double[] balances;
```

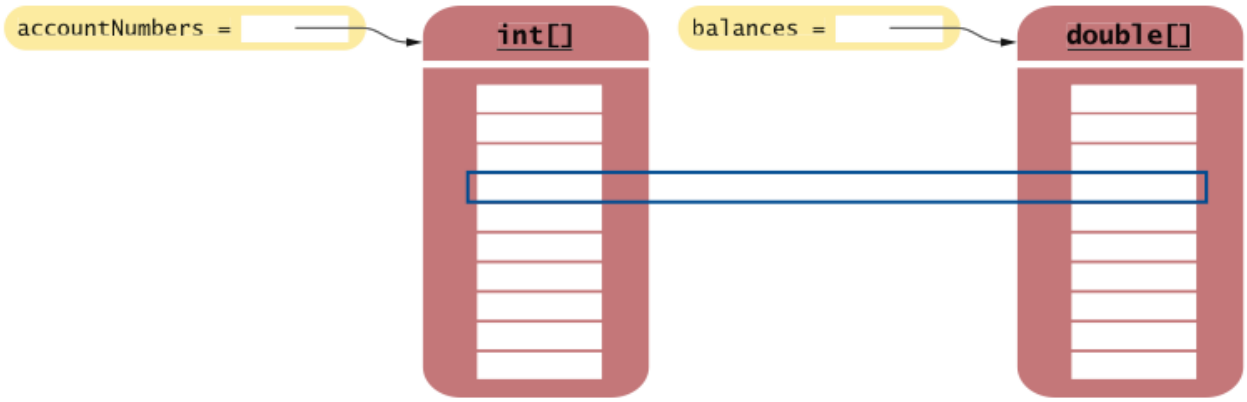


Figure 3 Avoid Parallel Arrays

Avoid parallel arrays by changing them into arrays of objects

```
BankAccount[] accounts;
```

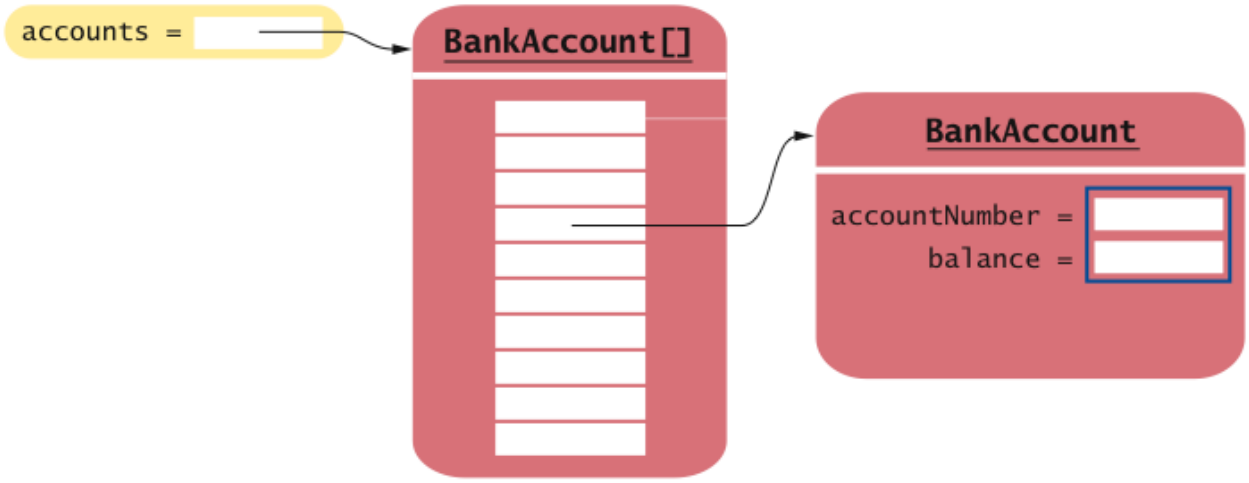
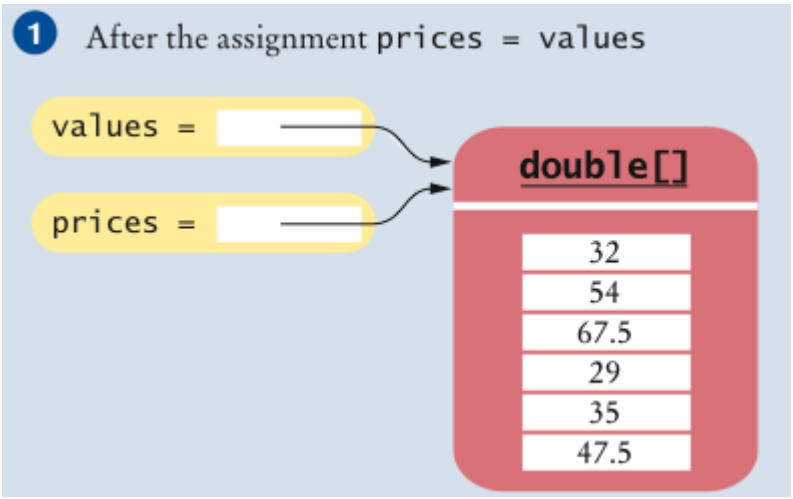


Figure 4 Reorganizing Parallel Arrays into an Array of Objects

Copying an array

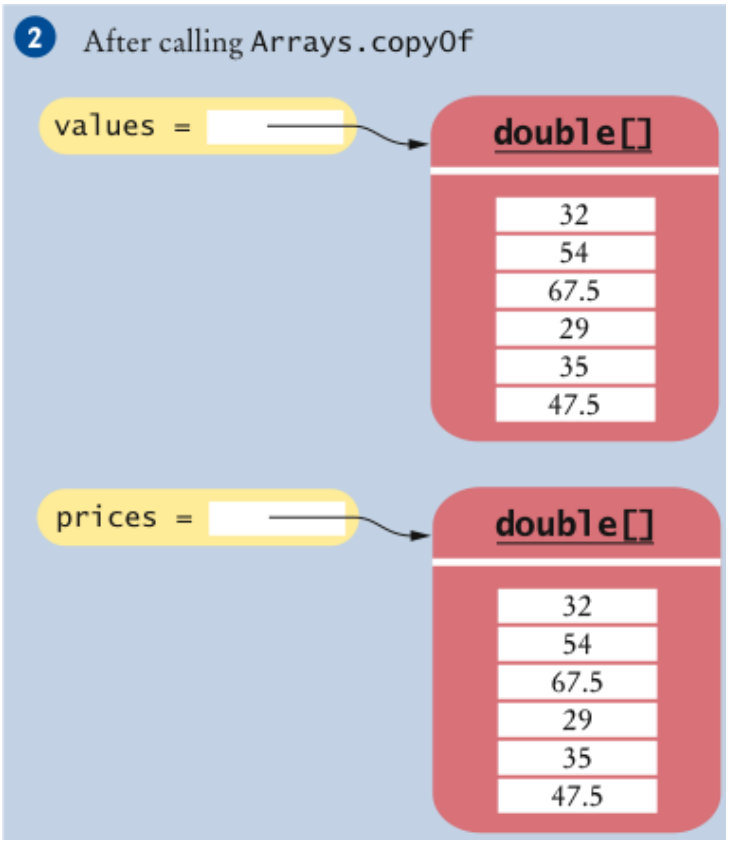
```
double[] values = new double[6];
... // statements to fill the array values[]

double[] prices = values;
//both values and prices refer to the same array object
```



To make a true copy of an array, call the **`copyOf`** method in the utility class **`Arrays`**

```
double[] prices = Arrays.copyOf(values, values.length);
```



To grow an array that has run out of space

```
values = Arrays.copyOf(values, 2 * values.length);
```

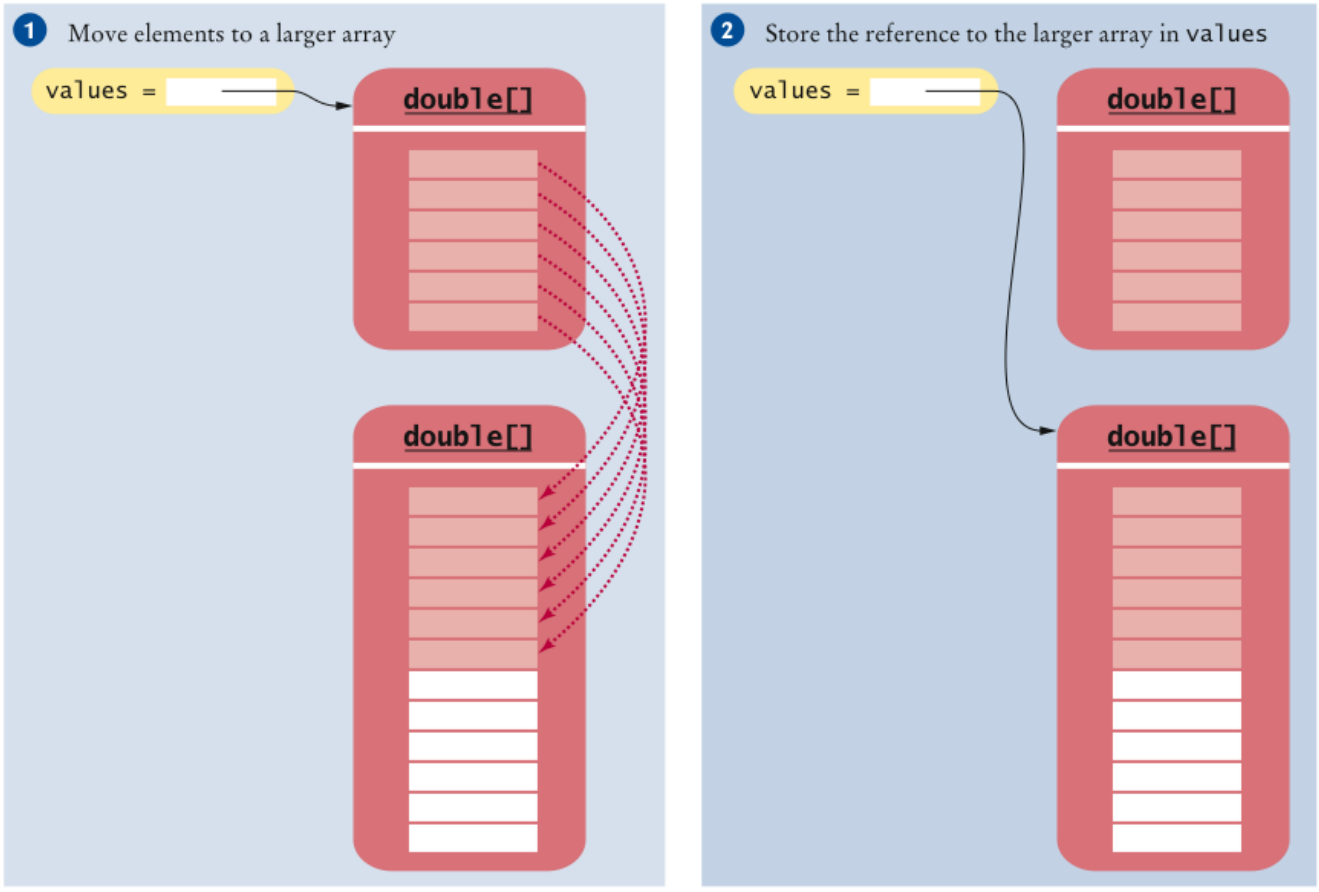


Figure 14 Growing an Array

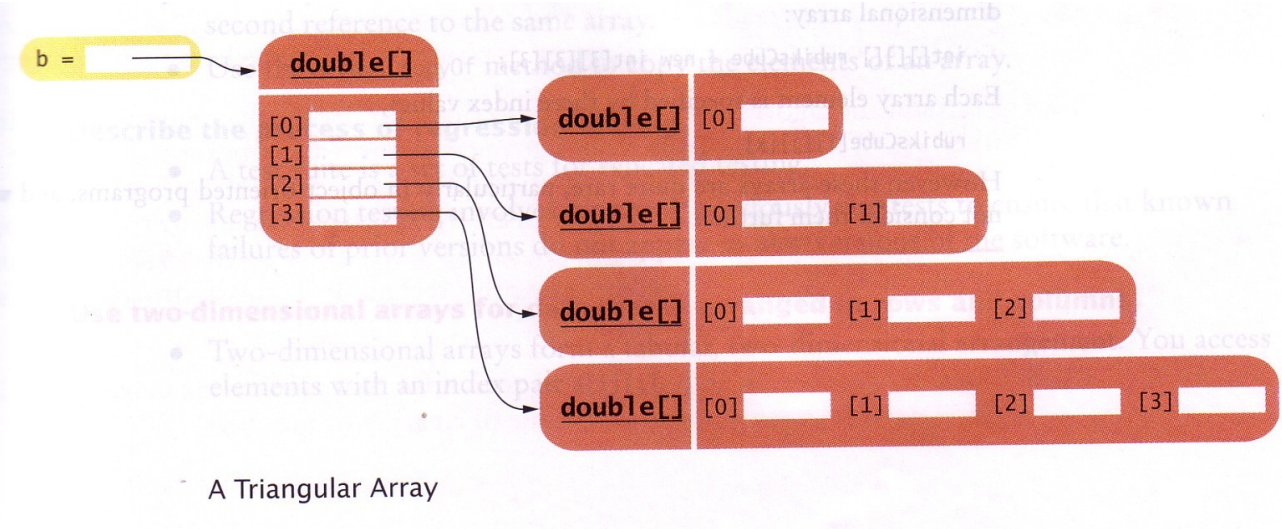
The **`class Arrays`** provides a number of useful utility functions for the manipulation of arrays, e.g. `copyOf()`, `sort()`, `binarySearch()`, etc.

Two dimensional (and higher dimensional) arrays

```
int[][] a = new int[3][4];
//2D-array with 3 rows, 4 columns


//create a triangular 2D-array
double[][] b = new double[3][];

for (int i = 0; i < b.length; i++)
   b[i] = new double[i+1];
```



A Triangular Array

In the above example, the number of rows in `b` is `b.length`, and length of the `i`-th row is `b[i].length`