Java™ Programming Language

SL-275-SE6

Module 1

Getting Started

What Is the Java™ Technology?

- Java technology is:
  - A programming language
  - A development environment
  - An application environment
  - A deployment environment
- It is similar in syntax to C++.
- It is used for developing both *applets* and *applications*.

Primary Goals of the Java Technology

The following features fulfill these goals:

- The Java Virtual Machine (JVM™)[1]
- Garbage collection
- The Java Runtime Environment (JRE)
- JVM tool interface

[1] The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform

The Java Virtual Machine

JVM provides definitions for the:

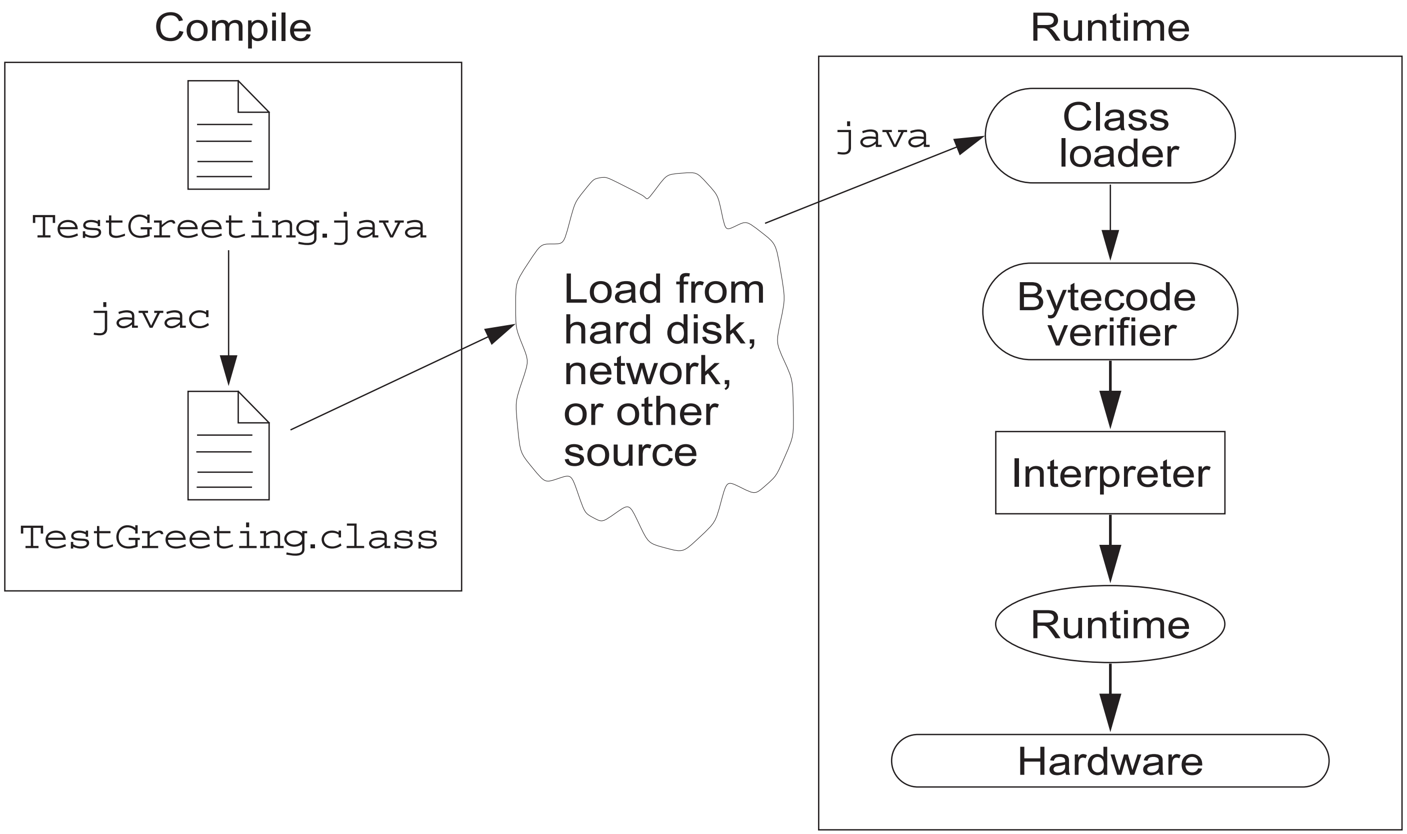
- Instruction set (central processing unit [CPU])
- Register set
- Class file format
- Stack
- Garbage-collected heap
- Memory area
- Fatal error reporting
- High-precision timing support

Garbage Collection

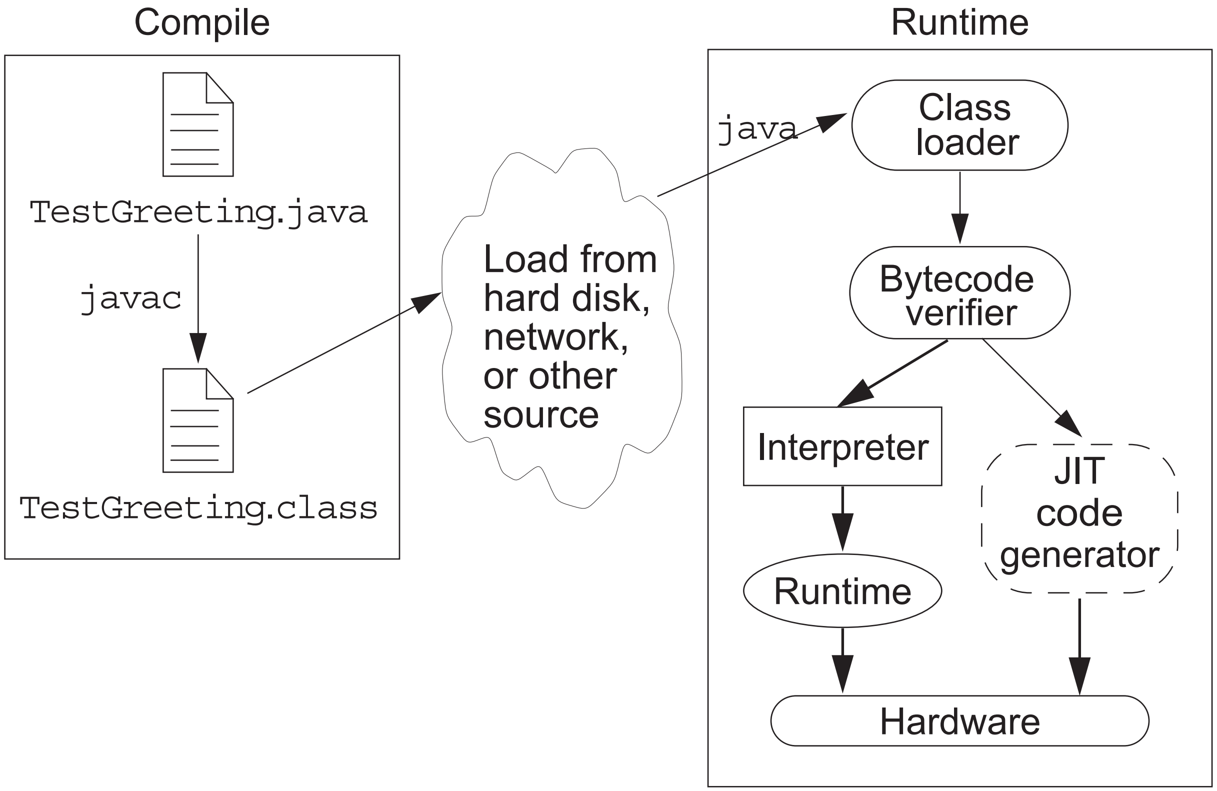
- Allocated memory that is no longer needed should be deallocated.
- In other languages, deallocation is the programmer’s responsibility.
- The Java programming language provides a system-level thread to track memory allocation.
- Garbage collection has the following characteristics:
  - Checks for and frees memory no longer needed
  - Is done automatically
  - Can vary dramatically across JVM implementations

The Java Runtime Environment

The Java application environment performs as follows:



Operation of the JRE With a Just-In-Time (JIT) Compiler



The Class Loader

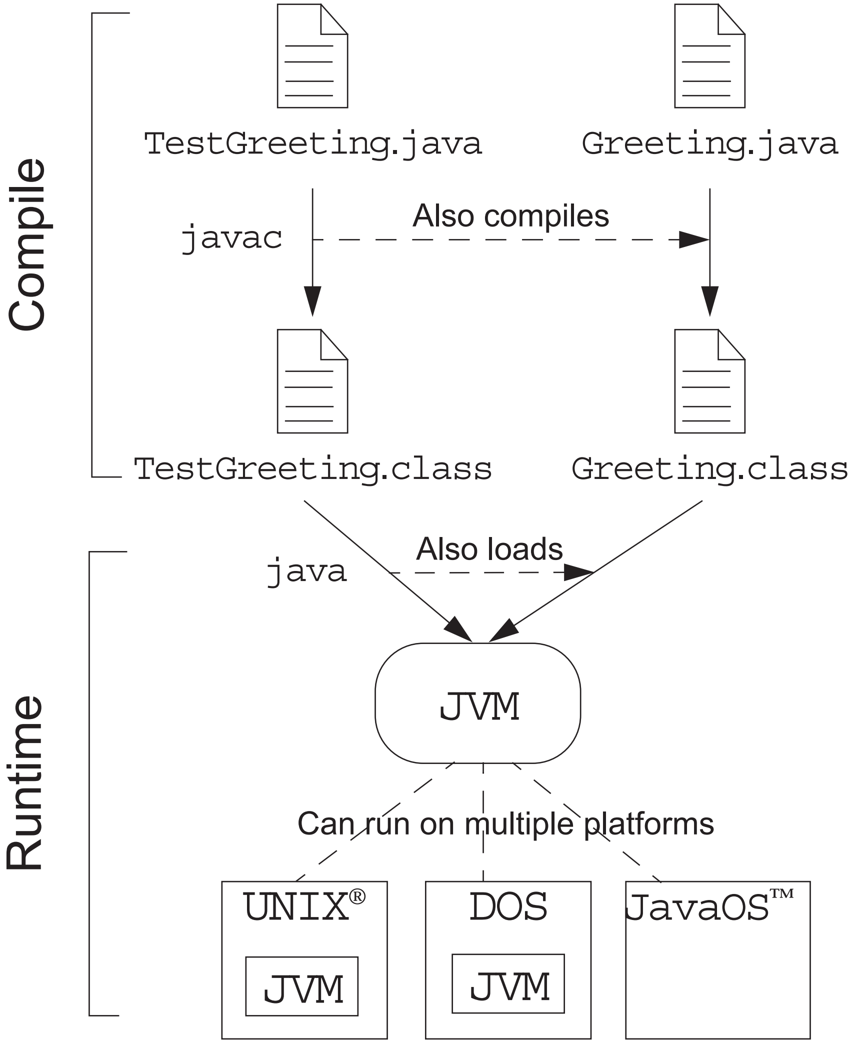
- Loads all classes necessary for the execution of a program
- Maintains classes of the local file system in separate *namespaces*
- Prevents spoofing

The Bytecode Verifier

Ensures that:

- The code adheres to the JVM specification.
- The code does not violate system integrity.
- The code causes no operand stack overflows or underflows.
- The parameter types for all operational code are correct.
- No illegal data conversions (the conversion of integers to pointers) have occurred.

Java Technology Runtime Environment



Module 2

Object-Oriented Programming

Classes as Blueprints for Objects

- In manufacturing, a blueprint describes a device from which many physical devices are constructed.
- In software, a class is a description of an object:
  - A class describes the data that each object includes.
  - A class describes the behaviors that each object exhibits.
- In Java technology, classes support three key features of object-oriented programming (OOP):
  - Encapsulation
  - Inheritance
  - Polymorphism

Declaring Java Technology Classes

- Basic syntax of a Java class:

```
<modifier>* class <class_name> {
  <attribute_declaration>*
  <constructor_declaration>*
  <method_declaration>*
}
```

- Example:

```
public class Vehicle {
  private double maxLoad;
  public void setMaxLoad(double value) {
    maxLoad = value;
  }
}
```

Declaring Attributes

- Basic syntax of an attribute:

```
<modifier>* <type> <name> [ = <initial_value>];
```

- Examples:

```
public class Foo {
  private int x;
  private float y = 10000.0F;
  private String name = "Bates Motel";
}
```

Declaring Methods

- Basic syntax of a method:

```
<modifier>* <return_type> <name> ( <argument>* ) {
  <statement>*
}
```

- Examples:

```
public class Dog {
  private int weight;
  public int getWeight() {
    return weight;
  }
  public void setWeight(int newWeight) {
    if ( newWeight > 0 ) {
      weight = newWeight;
    }
  }
}
```

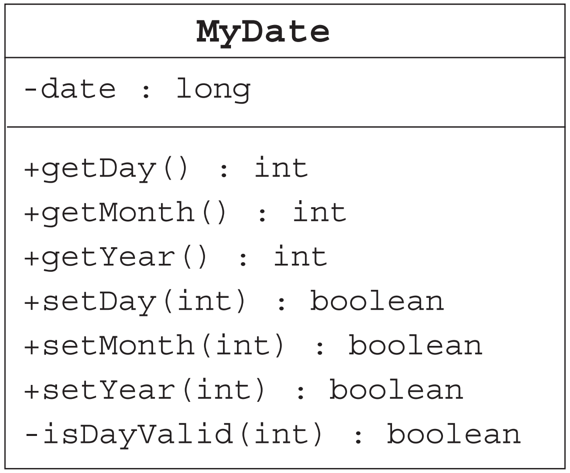
Accessing Object Members

- The *dot* notation is: **`<object>.<member>`**
- This is used to access object members, including attributes and methods.
- Examples of dot notation are:

```
d.setWeight(42);
d.weight = 42;  // only permissible if weight is public
```

Encapsulation

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code more maintainable



```
MyDate

-date : long

+getDay() : int
+getMonth() : int
+getYear() : int
+setDay(int) : boolean
+setMonth(int) : boolean
+setYear(int) : boolean
-isDayValid(int) : boolean
```

Declaring Constructors

- Basic syntax of a constructor:

```
[<modifier>] <class_name> ( <argument>* ) {
  <statement>*
}
```

- Example:

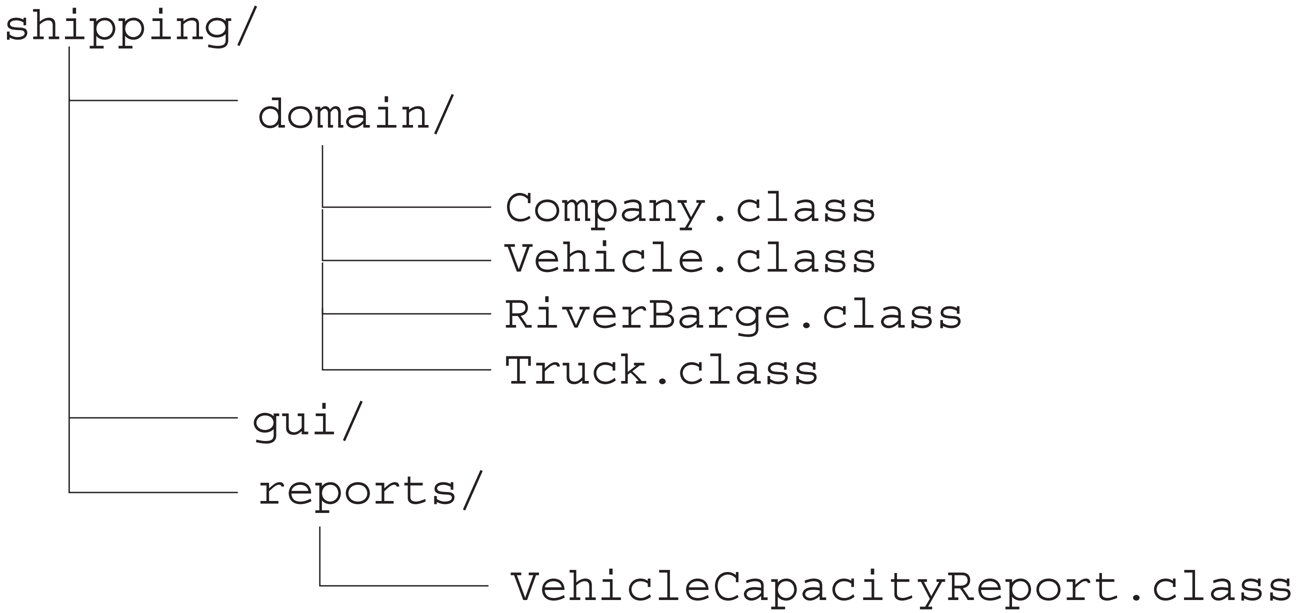
```
public class Dog {

  private int weight;

  public Dog() {
     weight = 42;
  }
}
```

The Default Constructor

- There is always at least one constructor in every class.
- If the writer does not supply any constructors, the default constructor is present automatically:
  - The default constructor takes no arguments
  - The default constructor body is empty
- The default enables you to create object instances with `new Xxx()` without having to write a constructor.

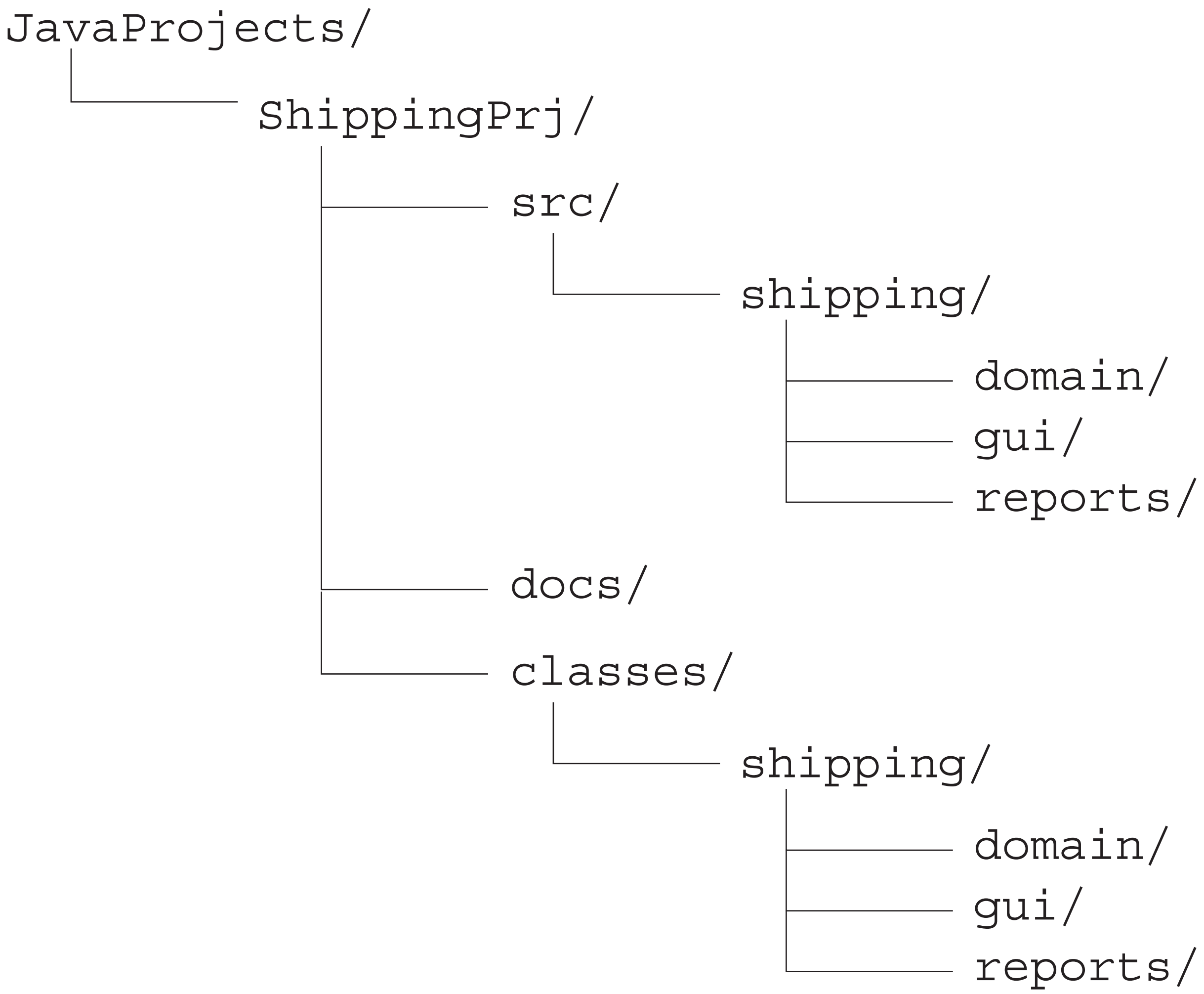
Directory Layout and Packages

- Packages are stored in the directory tree containing the package name.
- An example is the shipping application packages.



```
shipping/
   |
   |---- domain/
   |        |
   |        |---- Company.class
   |        |---- Vehicle.class
   |        |---- RiverBarge.class
   |        |---- Truck.class
   |---- gui/
   |---- reports/
            |
            |---- VehicleCapacityReport.class
```

Development



```
JavaProjects/
    ShippingPrj/
        src/
            shipping/
                domain/
                gui/
                reports/
        docs/
        classes/
            shipping/
                domain/
                gui/
                reports/
```

Compiling Using the `-d` Option

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

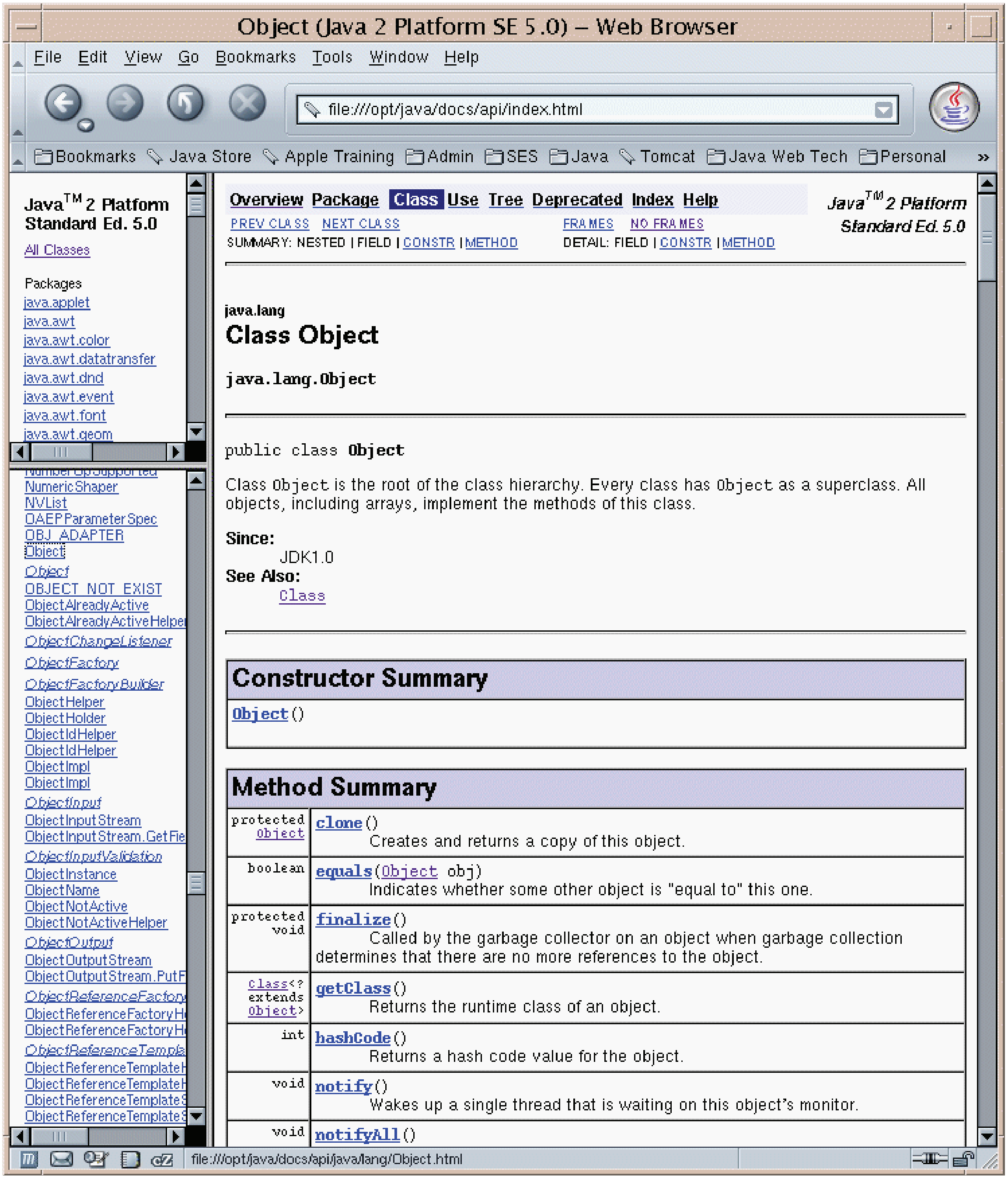
Terminology Recap

- Class – The source-code blueprint for a run-time object
- Object – An instance of a class;
  also known as *instance*
- Attribute – A data element of an object;
  also known as *data member*, *instance variable*, and *data field*
- Method – A behavioral element of an object;
  also known as *algorithm*, *function*, and *procedure*
- Constructor – A *method-like* construct used to initialize a new object
- Package – A grouping of classes and sub-packages

Using the Java Technology API Documentation

- A set of Hypertext Markup Language (HTML) files provides information about the API.
- A frame describes a package and contains hyperlinks to information describing each class in that package.
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on.

Java Technology API Documentation


Object (Java 2 Platform SE 5.0) – Web Browser
File Edit View Go Bookmarks Tools Window Help
file:///opt/java/docs/api/index.html
Bookmarks Java Store Apple Training Admin SES Java Tomcat Java Web Tech Personal
file:///opt/java/docs/api/java/lang/Object.html

Java™ 2 Platform Standard Ed. 5.0

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom

NumberUpSupported
NumericShaper
NVList
OAEPParameterSpec
OBJ_ADAPTER
Object
Object
OBJECT_NOT_EXIST
ObjectAlreadyActive
ObjectAlreadyActiveHelper
ObjectChangeListener
ObjectFactory
ObjectFactoryBuilder
ObjectHelper
ObjectHolder
ObjectIdHelper
ObjectIdHelper
ObjectImpl
ObjectImpl
ObjectInput
ObjectInputStream
ObjectInputStream.GetFie
ObjectInputValidation
ObjectInstance
ObjectName
ObjectNotActive
ObjectNotActiveHelper
ObjectOutput
ObjectOutputStream
ObjectOutputStream.PutF
ObjectReferenceFactory
ObjectReferenceFactoryH
ObjectReferenceFactoryH
ObjectReferenceTempla
ObjectReferenceTemplateH
ObjectReferenceTemplateH
ObjectReferenceTemplateS
ObjectReferenceTemplateS

Overview Package **Class** Use Tree Deprecated Index Help

Java™ 2 Platform Standard Ed. 5.0

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Class Object

`java.lang.Object`

`public class` **`Object`**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:
JDK1.0

See Also:
`Class`

Constructor Summary
`Object``()`

Method Summary	
`protected Object`	**`clone`**`()` Creates and returns a copy of this object.
`boolean`	**`equals`**`(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	**`finalize`**`()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<? extends Object>`	**`getClass`**`()` Returns the runtime class of an object.
`int`	**`hashCode`**`()` Returns a hash code value for the object.
`void`	**`notify`**`()` Wakes up a single thread that is waiting on this object's monitor.
`void`	**`notifyAll`**`()`

Module 3

Identifiers, Keywords, and Types

Identifiers

Identifiers have the following characteristics:

- Are names given to a variable, class, or method
- Can start with a Unicode letter, underscore (_), or dollar sign ($)
- Are case-sensitive and have no maximum length
- Examples:

```
identifier
userName
user_name
_sys_var1
$change
```

Java Programming Language Keywords

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

Reserved literal words: `null`, `true`, and `false`

Primitive Types

The Java programming language defines eight primitive types:

- Logical – `boolean`
- Textual – `char`
- Integral – `byte`, `short`, `int`, and `long`
- Floating – `double` and `float`

Module 4

Expressions and Flow Control

Variables and Scope

Local variables are:

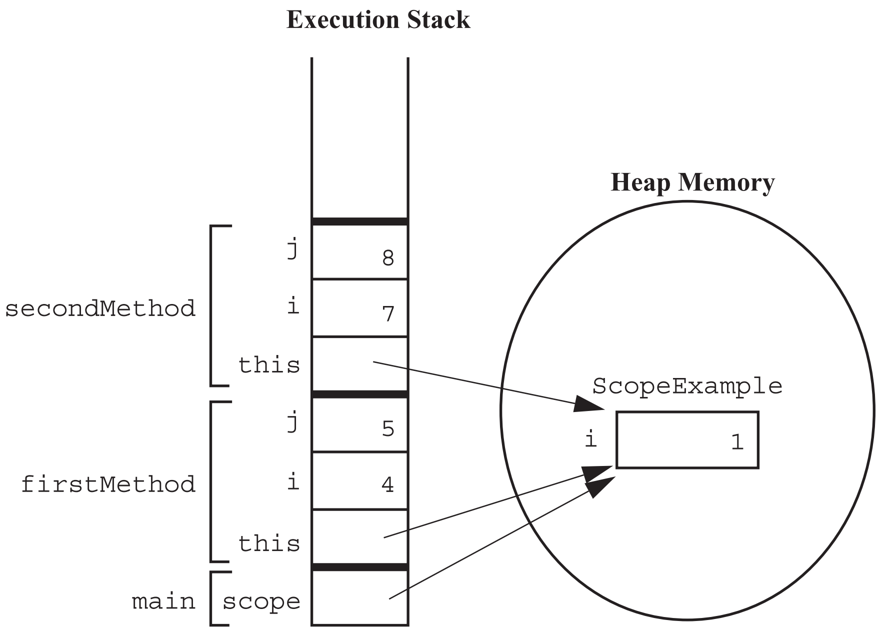
- Variables that are defined inside a method and are called *local*, *automatic*, *temporary,* or *stack* variables
- Variables that are created when the method is executed are destroyed when the method is exited

Variable initialization comprises the following:

- Local variables require explicit initialization.
- Instance variables are initialized automatically.

Variable Scope Example

```
public class ScopeExample {
  private int i=1;

  public void firstMethod() {
    int i=4, j=5;

    this.i = i + j;
    secondMethod(7);
  }
  public void secondMethod(int i) {
    int j=8;
    this.i = i + j;
  }
}


public class TestScoping {
  public static void main(String[] args) {
    ScopeExample scope = new ScopeExample();

    scope.firstMethod();
  }
}
```



Variable Initialization

Variable	Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0D`
`char`	`'\u0000'`
`boolean`	`false`
All reference types	`null`

Initialization Before Use Principle

The compiler will verify that local variables have been initialized before used.

```
3     public void doComputation() {
4       int x = (int)(Math.random() * 100);
5       int y;
6       int z;
7       if (x > 50) {
8         y = 9;
9       }
10      z = y + x;  // Possible use before initialization
11    }
```

```
javac TestInitBeforeUse.java
TestInitBeforeUse.java:10: variable y might not have been initialized
    z = y + x;  // Possible use before initialization
        ^

1 error
```

Operator Precedence

Operators	Associative
++ -- + *unary* - *unary* ~ ! (*<data_type>*)	R to L
* / %	L to R
+ -	L to R
<< >> >>>	L to R
< > <= >= instanceof	L to R
== !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
<boolean_expr> **?** *<expr1>* **:** *<expr2>*	R to L
= *= /= %= += -= <<= >>= >>>= &= ^= |=	R to L

String Concatenation With +

- The + operator works as follows:
  - Performs `String` concatenation
  - Produces a new `String`:

```
String salutation = "Dr.";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

- One argument must be a `String` object.
- Non-strings are converted to `String` objects automatically.

Casting

- If information might be lost in an assignment, the programmer must confirm the assignment with a cast.
- The assignment between `long` and `int` requires an explicit cast.

```
long bigValue = 99L;
int squashed = bigValue;       // Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L;            // Wrong, needs a cast
int squashed = (int) 99L;      // OK, but...
int squashed = 99;             // default integer literal
```

Promotion and Casting of Expressions

- Variables are promoted automatically to a longer form (such as `int` to `long`).
- Expression is *assignment-compatible* if the variable type is at least as large (the same number of bits) as the expression type.

```
long bigval = 6;      // 6 is an int type, OK
int smallval = 99L;   // 99L is a long, illegal

double z = 12.414F;   // 12.414F is float, OK
float z1 = 12.414;    // 12.414 is double, illegal
```

Simple `if, else` Statements

The `if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
if ( x < 10 )
  System.out.println("Are you finished yet?");
```

or (*recommended*):

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
}
```

Complex `if, else` Statements

The `if-else` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else
  <statement_or_block>
```

Example:

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
} else {
  System.out.println("Keep working...");
}
```

Complex `if`, `else` Statements

The `if-else-if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
int count = getCount(); // a method defined in the class
if (count < 0) {
  System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
  System.out.println("Error: count value is too big.");
} else {
  System.out.println("There will be " + count +
                     " people for lunch today.");
}
```

Switch Statements

The `switch` statement syntax:

```
switch ( <expression> ) {
  case <constant1>:
    <statement_or_block>*
    [break;]
  case <constant2>:
    <statement_or_block>*
    [break;]
  default:
    <statement_or_block>*
    [break;]
}
```

Switch Statements

A `switch` statement example:

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
    addRadio();
    addWheels();
    addEngine();
    break;
  case STANDARD:
    addRadio();
    addWheels();
    addEngine();
    break;
  default:
    addWheels();
    addEngine();
}
```

Switch Statements

This switch statement is equivalent to the previous example:

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
  case STANDARD:
    addRadio();
  default:
    addWheels();
    addEngine();
}
```

Without the break statements, the execution falls through each subsequent case clause.

Looping Statements

The `for` loop:

```
for ( <init_expr>; <test_expr>; <alter_expr> )
  <statement_or_block>
```

Example:

```
for ( int i = 0; i < 10; i++ )
  System.out.println(i + " squared is " + (i*i));
```

or (*recommended*):

```
for ( int i = 0; i < 10; i++ ) {
  System.out.println(i + " squared is " + (i*i));
}
```

Looping Statements

The `while` loop:

```
while ( <test_expr> )
  <statement_or_block>
```

Example:

```
int i = 0;
while ( i < 10 ) {
  System.out.println(i + " squared is " + (i*i));
  i++;
}
```

Looping Statements

The do/while loop:

```
do
  <statement_or_block>
while ( <test_expr> );
```

Example:

```
int i = 0;
do {
  System.out.println(i + " squared is " + (i*i));
  i++;
} while ( i < 10 );
```

Special Loop Flow Control

- The `break [<label>];` command
- The `continue [<label>];` command
- The `<label> : <statement>` command, where `<statement>` should be a loop

Module 5

Arrays

Declaring Arrays

- Group data objects of the same type.
- Declare arrays of primitive or class types:

```
char s[];
Point p[];

char[] s;
Point[] p;
```

- Create space for a reference.
- An array is an object; it is created with `new`.

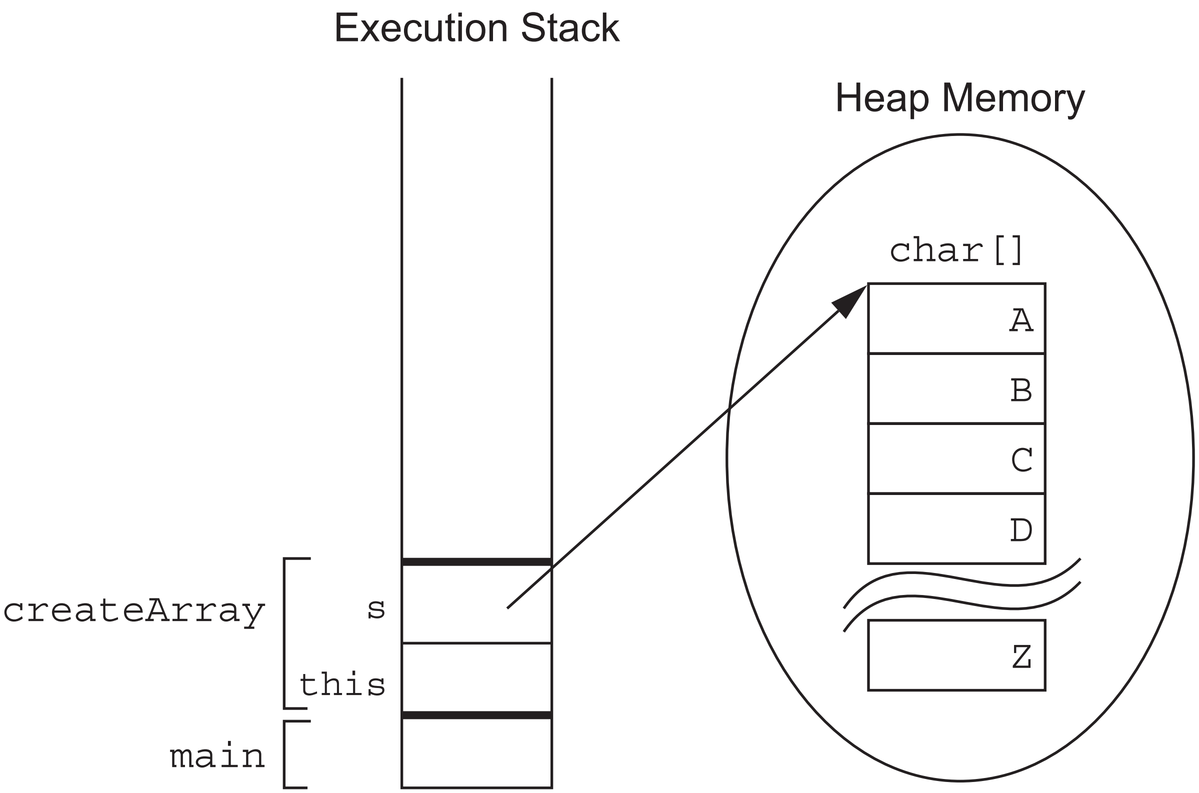
Creating Arrays

Use the `new` keyword to create an array object.

For example, a primitive (`char`) array:

```
public char[] createArray() {
  char[] s;

  s = new char[26];
  for ( int i=0; i<26; i++ ) {
    s[i] = (char) (’A’ + i);
  }

  return s;
}
```

Creating an Array of Character Primitives

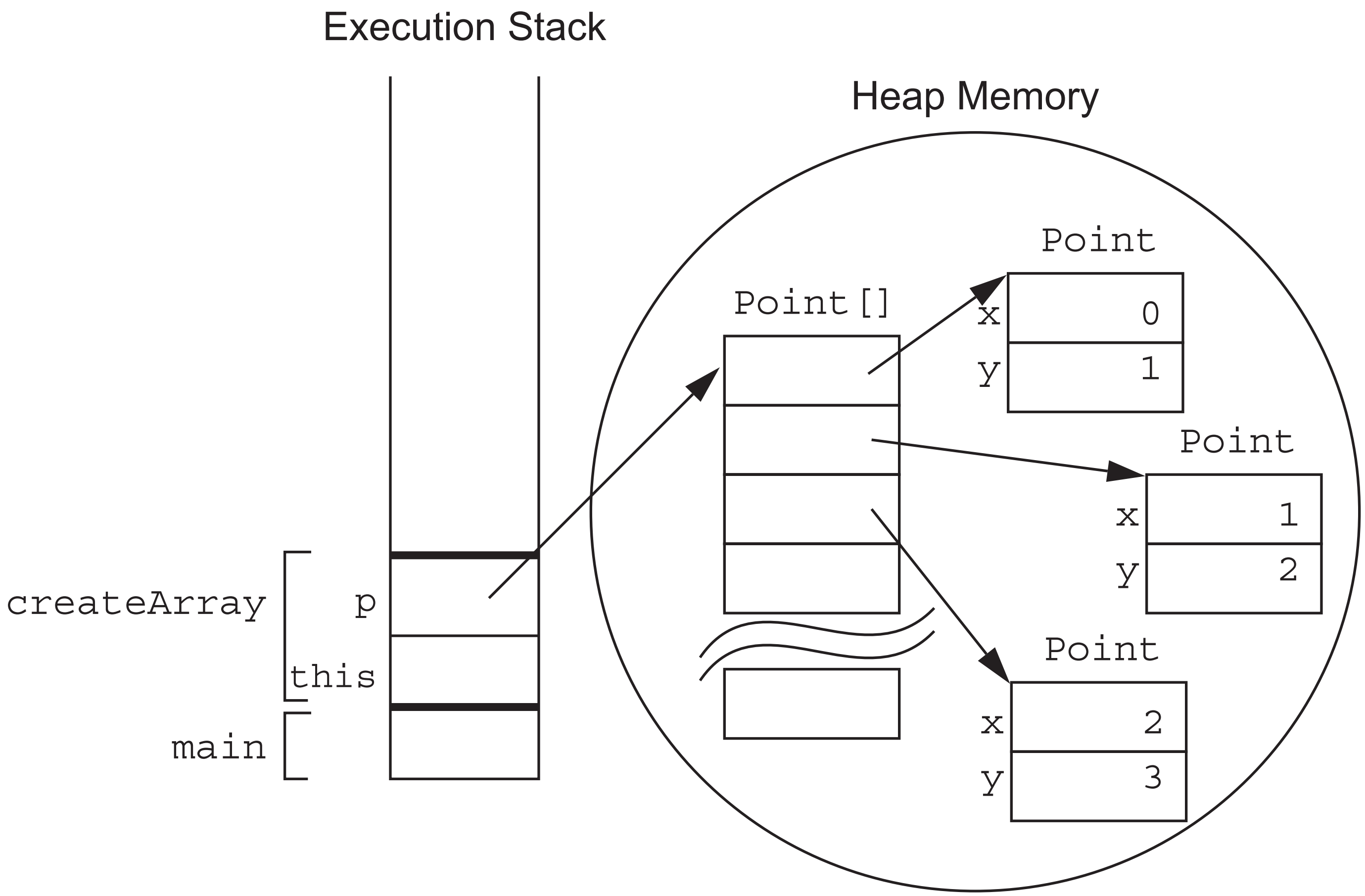


Creating Reference Arrays

Another example, an object array:

```
public Point[] createArray() {
  Point[] p;

  p = new Point[10];
  for ( int i=0; i<10; i++ ) {
    p[i] = new Point(i, i+1);
  }

  return p;
}
```

Creating an Array of Character Primitives With `Point` Objects



Initializing Arrays

- Initialize an array element.
- Create an array with initial values.

```
String[] names;
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

```
String[] names = {
    "Georgianna",
    "Jen",
    "Simon"
};
```

```
MyDate[] dates;
dates = new MyDate[3];
dates[0] = new MyDate(22, 7, 1964);
dates[1] = new MyDate(1, 1, 2000);
dates[2] = new MyDate(22, 12, 1964);
```

```
MyDate[] dates = {
    new MyDate(22, 7, 1964),
    new MyDate(1, 1, 2000),
    new MyDate(22, 12, 1964)
};
```

Multidimensional Arrays

Arrays of arrays:

```
int[][] twoDim = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];

int[][] twoDim = new int[][4]; // illegal
```

Multidimensional Arrays

- Non-rectangular arrays of arrays:

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

- Array of four arrays of five integers each:

```
int[][] twoDim = new int[4][5];
```

Array Bounds

All array subscripts begin at 0:

```
public void printElements(int[] list) {
  for (int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
  }
}
```

Using the Enhanced `for` Loop

Java 2 Platform, Standard Edition (J2SE™) version 5.0 introduced an enhanced `for` loop for iterating over arrays:

```
public void printElements(int[] list) {
  for ( int element : list ) {
    System.out.println(element);
  }
}
```

The `for` loop can be read as *for each* `element` *in* `list` *do*.

Array Resizing

- You cannot resize an array.
- You can use the same reference variable to refer to an entirely new array, such as:

```
int[] myArray = new int[6];
myArray = new int[10];
```

Copying Arrays

The `System.arraycopy()` method to copy arrays is:

```
//original array
int[] myArray = { 1, 2, 3, 4, 5, 6 };

// new larger array
int[] hold = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copy all of the myArray array to the hold
// array, starting with the 0th index
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```