Introduction to Java Programming

Features of Java

- Simple
- Secure
- Portable (platform-independent)
- Object-Oriented Programming
- Robust (memory and exception management)
- Multithreaded
- Distributed
- Dynamic

Object-oriented Programming Support

- 4 main features
  - Data Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Bytecode

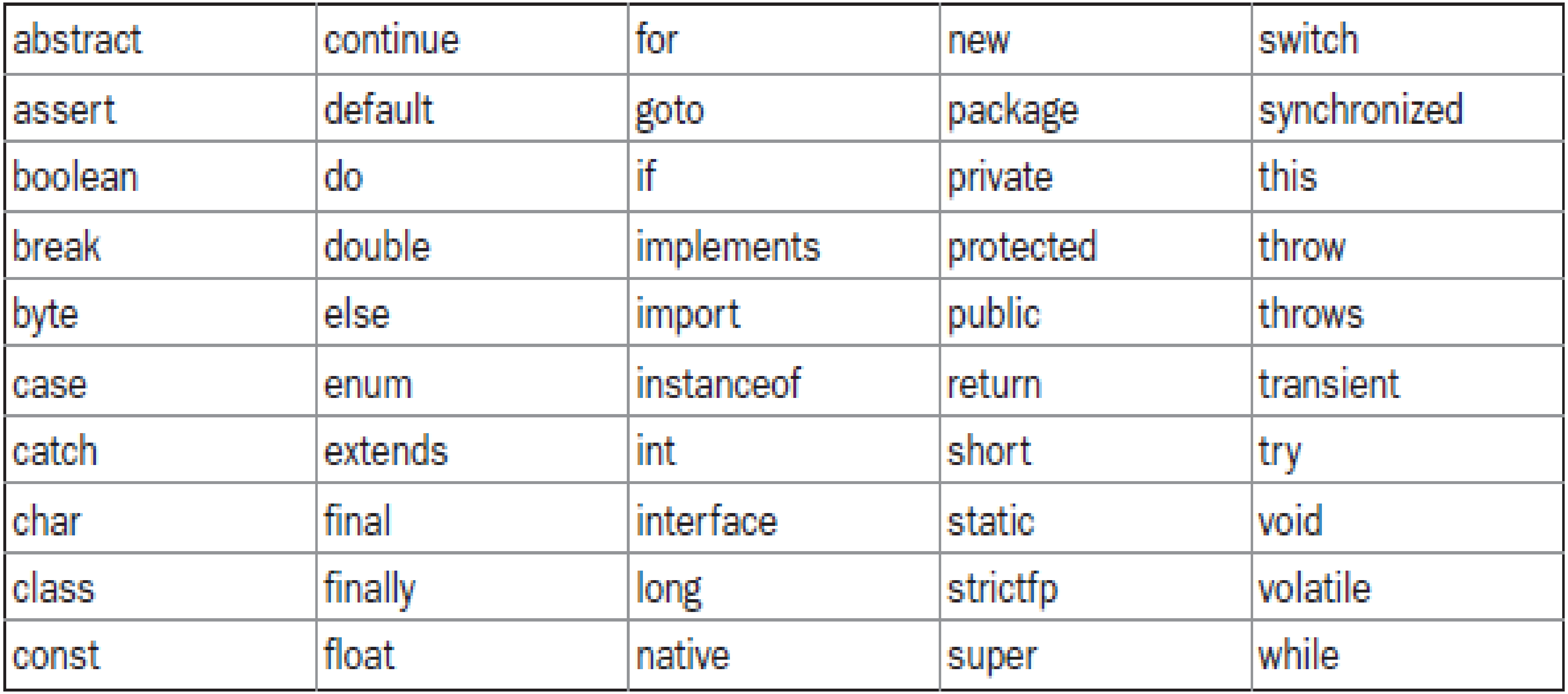
- It supports security and portability
- The output of a Java compiler is not executable code. Rather, it is bytecode.
- **Bytecode** is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- Why Bytecode
  - Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments.
  - Only the JVM needs to be implemented for each platform.
  - JVM will differ from platform to platform, but execute the same Java bytecode.

Simple Program Demonstration

```
/*
This is a simple Java program.
Call this file "FirstProg.java".
*/
class FirstProg {
// Your program begins with a call to main().
    public static void main(String args[]) {
        System.out.println("This is a simple Java program.");
    }
}
```

Java Keywords

- 50 keywords



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

Datatypes

- Java defines 8 primitive data types
- **byte, short, int, long, char, float, double** and **boolean**
- The primitive types represent single values, not complex objects
- Grouped into 4 categories
  - Integers : **byte, short, int,** and **long**
  - Floating-point numbers : **float** and **double**
  - Characters : **char**
  - Boolean : **boolean**

Integers

- Java does not support unsigned, positive-only integers
- The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them
- long
  - 64 bits
  - Range : -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- int
  - 32 bits
  - Range: -2,147,483,648 to 2,147,483,647
- short
  - 16 bits
  - Range: -32,768 to 32,767
- byte
  - 8 bits
  - Range: -128 to 127

A program with long data type

```
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

Floating-Point

- Also known as real numbers
- Used for fractional precision
- double
  - 64 bits
  - Range: 4.9e–324 to 1.8e+308
- float
  - 32 bits
  - Range: 1.4e–045 to 3.4e+038
- In some modern processor, double precisions are faster than float

A program with floating points

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area

    System.out.println("Area of circle is " + a);
  }
}
```

characters

- Java uses Unicode to represent characters.
- Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- Thus, in Java char is a 16-bit type and the range is 0 to 65,536.
- There are no negative chars.
- The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255

Examples for char data types

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

```
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

Booleans

- Defined using boolean keywords
- Takes either **false** or **true** as literals

```
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

Dynamic initialization of variables

- Use variable whenever required
- Variables can be initialized dynamically, using any valid expression at the time of declaration

```
class DynInit {
  public static void main(String args[]) {
   double a = 3.0, b = 4.0;

   // c is dynamically initialized
   double c = Math.sqrt(a * a + b * b);

   System.out.println("Hypotenuse is " + c);
  }
}
```

The Scope and Lifetime of Variables

- Java allows variables to be declared within any block
- A block is begins and ends with { and } respectively.
- A block defines a scope
- Variables declared inside a scope are not visible to code that is defined outside that scope.
- Its a way to localizing a variable and protecting it from unauthorized access and/or modification.
- Provide the foundation for encapsulation.
- Scopes can be nested

Demonstration of block scope

```
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}

class Scope{
   public static void main(String args[]) {
     int bar = 1;
     {              // creates a new scope
       int bar = 2;
     }
   }
}
```

Type Conversion and Casting

- Automatic conversion for compatible types
- For incompatible types, explicit conversion via casting is required
- Java’s Automatic Conversions will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a widening conversion takes place
- Casting Incompatible types is called a narrowing conversion
- Casting has this general form: (target-type) value

An example

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

Automatic Type Promotion in Expressions

- Consider the following code

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

What is the type of sub expression a*b?

The Type Promotion Rules

- First, all **byte**, **short**, and **char** values are promoted to **int**.
- If one operand is a **long**, the whole expression is promoted to **long**.
- If one operand is a **float**, the entire expression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

An example

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

One-Dimensional Arrays

- The general form of a one-dimensional array declaration is
  - type var-name[ ];
- With this array variable is set to null
- We can allocate memory to array by using **new** operator
  - array-var = new type[size];
  - Further, all elements in the array will be initialized to zero.

1D array example

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + "
days.");
  }
}
```

1D array example(2)

```
class AutoArray {
  public static void main(String args[]) {
    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

Two dimensional array

```
class TwoDArray {
  public static void main(String args[]) {
    int twoD[ ][ ]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

Two dimensional array

- Manually allocating size of second dimention

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- Another way of initialization of 2D array

```
double m[][] = {
  { 0*0, 1*0, 2*0, 3*0 },
  { 0*1, 1*1, 2*1, 3*1 },
  { 0*2, 1*2, 2*2, 3*2 },
  { 0*3, 1*3, 2*3, 3*3 }
};
```

Alternative Array Declaration Syntax

- The following two declarations are equivalent:
  - int a1[] = new int[3];
  - int[] a2 = new int[3];
- The following declarations are also equivalent:
  - char twod1[][] = new char[3][4];
  - char[][] twod2 = new char[3][4];
- For example,
  - int[] nums, nums2, nums3; // create three arrays