

# Introduction to *Java*

- James Gosling, Sun Microsystems
  - Originally given name Oak.
  - 1995 - Java 1.0
  - 1997 - Java 1.1
  - 1999 - Java 1.2
  - Java 1.3 - soon ...
- 
- *A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, dynamic language.*

# Java - *Language Features*

- **Object-oriented**

- Programs written in terms of classes.
- Extensive set of pre-defined classes, arranged in packages:

java.awt

Primitive GUI classes

java.io

Input/output classes

...

- **Interpreted**

- Java compiler generates byte codes for the Java virtual machine (rather than native machine code).
- Interpreter used to execute compiled byte codes.
  - *architecture neutral*

- **Dynamic**

- A Java class can be loaded into a running Java interpreter at any time.

# Java - *Language Features*

- **Distributed**

- Extensive language support for networking.
  - e.g. reading remote files/resources.

- **Robust**

- Language strongly typed.
- No pointers/pointer arithmetic.
- Automatic garbage collection
  - programmer relieved of memory management tasks.
- Exception handling
  - error checking/handling

- **Performance**

- Java1.1 approx. 10 times slower than C.
- Java interpreters include *just in time* compilers which translate byte codes into machine code for particular CPU at run-time.

# Java - *Language Features*

- **Secure**

- Distributed nature of Java requires a security model.
- Defence against malicious code
  - lack of pointers, inability to read memory outside bounds of arrays/strings.
- Byte code verification performed by interpreter.
- Untrusted code & SecurityManager
  - restrictions imposed on code
  - e.g. access to local file system
- Digital signature to establish origin of code.

- **Multi-threaded**

- Support for multiple threads of execution (lightweight processes) that can handle different tasks.

# Java - *Applications & Applets*

- **Applications**

- Stand-alone programs.
- Consist of one or more class definitions.
  - One contains a `main( )` method - program execution commences here.
  - `main( )` method has following prototype:

```
public static void main(String[] args)
```
- To start program: invoke Java interpreter on class containing `main( )` method.

- **Applets**

- Small, lightweight applications; run within Web browsers or other "applet viewers".
- Security restrictions apply on untrusted code.
- No `main( )` method.
- Created by defining a subclass of the existing `Applet` class and overriding a number of methods.

# Variables & Types

- **Variable names**
  - Legal Java identifier made up of a series of Unicode characters - letters, digits, underscores (\_), dollar signs (\$).
  - Must not begin with a digit.
  - Must not be same as keyword or literal.
  - Must not have same name as other variable whose declaration appears in same scope.
- **Conventions**
  - Variable names start with lowercase.
  - Class names start with Uppercase.

# Primitive Data Types

Type	Size/Format	Description
byte	8 bit 2's complement	Byte length integer
short	16 bit 2's complement	Short integer
int	32 bit 2's complement	Integer
long	64 bit 2's complement	Long integer
float	32 bit	Single precision float
double	64 bit	Double precision float
char	16 bit Unicode character	Single character
boolean	true/false	Boolean value

- No unsigned keyword as in C.
- boolean values not integers; may not be treated as integers.
- Strings in Java not a primitive type; instances of String class.
- ```
char c = 'A';  
String str = "Foo";
```

# Operators

- **Arithmetic**

$+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$

- **String concatenation**

$+$

- **Relational/conditional**

$<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$ ,  $!=$ ,  $\&\&$ ,  $||$

- **Incremental**

$++$ ,  $--$

- **Assignment**

$=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $+=$ ,  $-=$ ,  $\dots$

- **Bitwise**

$<<$ ,  $>>$ ,  $>>>$ ,  $\sim$ ,  $\&$ ,  $\wedge$ ,  $|$

- **Boolean**

$!$ ,  $\&$ ,  $\wedge$ ,  $|$

- Standard operators have same precedence as in C.



# Operators

| Operator | Use                             | Operation                                                                    |
|----------|---------------------------------|------------------------------------------------------------------------------|
| >>       | <code>op1&gt;&gt;op2</code>     | Shift bits of <code>op1</code> right by distance <code>op2</code>            |
| <<       | <code>op1&lt;&lt;op2</code>     | Shift bits of <code>op1</code> left by distance <code>op2</code>             |
| >>>      | <code>op1&gt;&gt;&gt;op2</code> | Shift bits of <code>op1</code> right by distance <code>op2</code> (unsigned) |
| &        | <code>op1&amp;op2</code>        | Bitwise AND                                                                  |
|          | <code>op1 op2</code>            | Bitwise OR                                                                   |
| ^        | <code>op1^op2</code>            | Bitwise XOR                                                                  |
| ~        | <code>~op1</code>               | Bitwise complement                                                           |

# Control Statements

- Similar (or identical) to C statements.

| Statement        | Keyword(s)                      |
|------------------|---------------------------------|
| <i>Decision</i>  | if/else, switch/case/default    |
| <i>Loop</i>      | for, while, do/while            |
| <i>Exception</i> | try/catch/finally, throw        |
| <i>Misc.</i>     | break, continue, label:, return |

- Compound statements created by enclosing in curly braces - { }

# Decisions

- `if/else`

```
if (boolean expression)
    statement1
else
    statement2
```

```
if ((i%5) == 0)
    System.out.print("Foo!");
```

- In Java, the values `0` and `null` are not the same as `false`; non-zero and non-null values not the same as `true`.

# Decisions

- switch/case/default

```
switch (expression) {  
    case i: ...; break;  
    case j: ...; break;  
    default: ...  
}
```

```
switch (a % 5) {  
    case 0:  
        System.out.print("Foo!");  
        break;  
    case 1:  
        System.out.print("Bar!");  
        break;  
    default:  
        System.out.print("Eek.");  
}
```

- Values in case statements must be constants, and of integral types (byte, char, short, int, long)

# Loops

- for

```
for (initialise; test; update)
    body
```

```
for (int i=1; i <= 100; i++) {
    System.out.print("Foo!");
}
```

- Multiple, comma-separated expressions allowed in initialise and update sections - not test.
- Local variables can be declared in initialise section.

# Loops

- while

```
while (boolean expression)
    body
```

```
while (i < 10) {
    System.out.print("Foo!");
    i++;
}
```

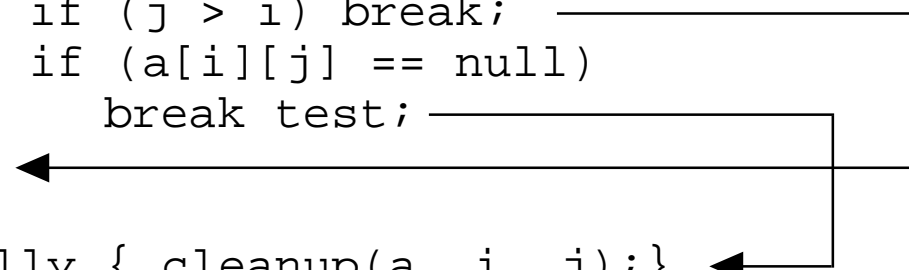
- do/while

```
do body
    while (boolean expression)
```

# Branching

- `break`
  - Exits from current statement.
  - `break <label>;`
    - Used to break from specified statement/loop not just nearest enclosing one.
  - Example:

```
test: if (foo(i)) {  
    try {  
        for (int j=0; j <10; j++) {  
            if (j > i) break;  
            if (a[i][j] == null)  
                break test;  
        }  
    }  
    finally { cleanup(a, i, j); }  
}
```



# Branching

- `continue`
  - Exits from current loop.
  - `continue <label>;`
    - Applies to any enclosing loop (not just nearest enclosing loop).
    - Example:

```
outer: while (!done) {  
    if test(a, b) == 0 continue;  
    try {  
        for (int i=0; i<10; i++) {  
            if (a[i] == null)  
                continue;  
            else if (b[i] == null)  
                continue outer;  
        }  
    }  
    finally { cleanup(a, b); }  
}
```

```
graph TD  
    C1[continue] --> Try[try {  
    for (int i=0; i<10; i++) {  
        if (a[i] == null)  
            continue;  
        else if (b[i] == null)  
            continue outer;  
    }  
}]  
    C2[continue outer] --> While[while (!done) {  
    if test(a, b) == 0 continue;  
    try {  
        for (int i=0; i<10; i++) {  
            if (a[i] == null)  
                continue;  
            else if (b[i] == null)  
                continue outer;  
        }  
    }  
    finally { cleanup(a, b); }  
}]  
    F[cleanup(a, b);] --> While
```



# return

- Exits from current method and returns to calling method:

```
return expression;
```

- Cannot use return to return exit value from `main()` method.
- Use `System.exit()` method with desired integer value.

# Arrays

- Manipulated by reference.
- Dynamically created using `new` keyword.
- Garbage collected when no longer needed.

- Creation:

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

- Array elements initialised to default value for type.
  - `int` - default is 0
  - Array of objects - default is `null`
- Creation with initialiser:

```
int collection[] = {1, 2, 4, 8, 16, 32};
```

- Elements in initialiser may be arbitrary expressions.

# Arrays

- Accessing array elements:

```
int a[] = new int[100];  
int i = 5;  
a[i] = 1;
```

- Length of array:

```
int a[] = new int[100];  
a.length
```

# Multi-dimensional Arrays

- Specify multi-dimensional array using appropriate number of `[]` pairs.
- Example:

```
int twodim[][] = new int[10][100];
```

- Allocates array with 10 elements, each of which is of type `int[]`.
- Can allocate and initialise with nested initialisations:

```
int param[][] = {  
    {1, 2, 3},  
    {101, 102, 103}  
};
```

# Strings

- **Not** null-terminated array of chars as in C!
- Instances of the `java.lang.String` class.
- `String` objects are immutable
  - No methods defined to change `String` contents.
  - `StringBuffer` object contents can be modified.
- `String` handling methods:
  - `length()`, `equals()`, `substring()` ...

# Command-Line Arguments

- Single argument to `main( )` method is array of strings (conventionally called `args`).
- Elements of array are the command-line arguments.
- Length of array:

`args.length`

- First element of array is **not** name of program (as C programmer would expect).

# Constants

- `final` keyword used to declare a variable constant.
- Value specified at declaration time.
- Within a class definition, include statement such as:

```
public static final double PI = 3.14159;
```

# Miscellaneous Differences with C

- No pointers.
  - Cannot cast object/array references into integers and vice-versa.
- No preprocessor
  - No `#include`, `#define` ...
- No C-like macros
- No `struct`, `union`
- No enumerated types (no `enum`)
- No `typedef`
- No variable length argument lists.
- Automatic garbage collection
  - objects created with `new` keyword
  - no corresponding `delete` or `free()` operation



# Structure of Java Source Code Files

- Filename ends with extension: .java
- Components of a source code file:

```
package <package name>;  
import <package name or wildcard>;  
  
public class <class name> {  
    <class body>  
}
```

Other class definitions...

- package
  - Indicates package to which source code belongs
  - package name - dot separated list:
    - pedwards.games.tetris
  - Compiled class file stored in directory with same structure as package name.
    - pedwards/games/tetris/Effects.class

# Structure of Java Source Code Files

- `package`
  - Java code in a package has access to all classes (public & non-public) in that package.
- `import`
  - Makes classes available to current class.
  - `import java.awt.*;`
    - All classes in `awt` package available.
- If more than one class defined in a file, only one may be declared `public`.
- Source file must have same name as `public` class.
- Java comments:
  - `/* . . . . */` C style (cannot be nested)
  - `//` Single line comment (C++ style)
  - `/** . . . . */` JavaDoc comment
    - JavaDoc comments processed by `javadoc` program.