# Java Programming

# Why Java?

- Platform independence
- Object Oriented design
- Run-time checks (fewer bugs)
- Exception handling
- Garbage collection
- some drawbacks (speed, RAM)

# Object Oriented Design

- software is built from objects
- each object is an instance of a class
- classes are arranged in a hierarchy
- each class defines fields and methods

# Types

- basic (primitive) types
  - boolean (true/false)
  - char (16 bits), byte (8 bits)
  - short (16), int (32), long (64)
  - float (32 bits), double (64 bits)
- Objects
  - "structured" types
  - must be "instantiated"

# Instantiation

- basic types (space allocated automatically)

  ```
  int n;
  ```

- arrays (must allocate space explicitly)

  ```
  int[] col = new int[5];
  ```

- Objects

  ```
  Gnome  g;                    (allocates reference only)
  g = new Gnome(...);  (calls constructor method)
  ```

  - several constructors may be defined for the same class, with different parameters.

# arrays

- array of Objects = array of "references"
  - two-step allocation

```
Gnome[] party = new Gnome[5];
for( int i=0; i < party.length; i++ ) {
        party[i] = new Gnome(...);        }
```

- 2-dimensional array = "array of arrays"

```
int table[][] = new int[10][20];
```

- triangular arrays (allocate each row separately)

```
int[][] table = new int[10][];
for( int i=0; i < table.length; i++ ) {
        table[i] = new int[i];
}
```

# Number Objects ("wrappers")

- basic type "wrapped" inside an Object

```
Integer N = new Integer(1045);
int n = N.intValue();

Double X = new Double( 3.934 );
double x = X.doubleValue();
```

- these classes provide useful methods

```
Integer.parseInt( String s )
Double.isInfinite( double v )
```

# String class

- concatenation

  String s = “kilo” + “meters”

- several useful methods

  int length()

  char charAt( int index )

  int compareTo( String other )

- see also StringBuffer

# Methods

- return types
- parameters
- constructor methods
- main method
- local variables

# Point class

```
public class Point
{
        private double x;
        private double y;

        public double getX() { return( x ); }
        public double getY() { return( y ); }

        public void translate( double dx, double dy ) {
                x += dx;
                y += dy;
        }
}
```

# Modifiers (class/field/method)

- Special purpose
  - static (all instances share same value)
  - final (can't reassign/override/subclass)
  - abstract (can't instantiate)
- Access control
  - public (anyone can access)
  - protected (same package or subclass)
  - friendly (default - only same package)
  - private (only same class can access)

# Expressions

- Literals
  - null
  - true, false
  - integer 42, 176L, -52I
  - Floating point 3.14159
  - Character '\t'
  - String "status quo ante"

# Operator Precedence

- ++ -- ~ ! (postfix, prefix, type conversion)
- * / % (multiply/divide)
- + - (add/subtract)
- << >> >>> (shift)
- < <= > >= (comparison) [also instanceof]
- == != (equality)
- & (bitwise-and)
- ^ (bitwise-xor)
- | (bitwise-or)
- && (and)
- || (or)
- ? : (conditional)
- = += -= *= /= %= >>= <<= >>>= &= ^= |= (assignment)

# Casting

- Numberical conversion

```
int ifrac, i=3, j=4;
double dfrac, d=3.2;

dfrac = i / d; //  ( i automatically cast to double)
dfrac = i / (double) j ; //  ( i again cast to double)
ifrac = i / d;       //  (will cause a compilation error)
ifrac = i / (int) d;//(loss of precision made explicit)
```

- String conversion

```
String s = “” + 22; // (correct, but ugly)
String s = Integer.toString(22); // (much better)
```

# Control Flow

- if( ) { ... } else { ... }
- switch
- loops
  - for( ; ; ) { ... }
  - while( ) { ... }
  - do { ... } while( );
- return
- break / continue

# Input / Output

- Output - very simple

  System.out.println("size:\t" + x );

- Input - a bit more complicated

  BufferedReader

  InputStreamReader

  System.in

  (see, for example Copy.java)

# Packages

- classes can be collected in a package
- standard packages are provided, or you can create your own
- import a single class

```
import java.io.InputStream;
```

- import an entire package

```
import java.io.*;
```

# Next Time

- Object Oriented design
- Inheritance/Polymorphism
- Exception handling
- Interfaces & abstract classes
- Casting
- Design Patterns
- Javadoc