

## A Complete Java Program

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
- javac HelloWorld.java
- java HelloWorld
```

## (Non-)Tokens

- **Whitespace**
  - blank
  - tab
  - newline
- **Comment**
  - // ...
  - /\* ... \*/
  - /\*\* ... \*/
- **Separators**
  - (, ), [, ], etc
- **Keywords (*reserved*)**
- **Identifiers**
- **Literals**
  - 25, 2.5 (*int/float*)
    - 09 is not an integer.
  - true, false (*boolean*)
  - 'a', \u0061, \n (*char*)
- **Operators**
  - +, >, &&, etc

## Java : Types

- Primitive Types
- Reference Types

## Primitive Types and Values

- **Numeric Types**
  - **Integral Types** (signed two's complement)
    - byte, short (2 bytes), int (4 bytes), long (8 bytes)
    - char (**16-bit Unicode**)
  - **Floating-point Types** (IEEE 754 Standard)
    - float (4 bytes), double (8 bytes)
- **Boolean Type**
  - boolean
    - true, false

## Numeric Types

- Explicit range and behavior.
  - Java trades performance for cross-platform portability.
- **byte** : appropriate when parsing a network protocol or file format, to resolve “endianess”.
  - BIG endian : SPARC, Power PC : MSB-LSB
  - LITTLE endian : Intel X86 : LSB-MSB
- Calculations involving **byte/short** done by promoting them to **int**.
- Internally, Java may even store **byte/short** as **int**, *except in arrays*.

## Type Conversions

- Coercion (widening or promotion)
  - `int i = 50; float f = i;`
- Fractional literals are always **double**.
- Casting (narrowing)
  - `byte b = 128; // Error`
  - `byte b = (byte) 258; // = 2`
  - `b = (b * 0); // Error`
  - `b = (byte) (b * 2);`
  - `char four = (char) ( '1' + 3 );`
  - `float f = (float) 0.0;`

## Casting

```
class PrimCast {
    public static void main(String[] argv) {
        byte b = 0;
        int i = b;
        b = i;           // *error*
        b = (b * 0);     // *error*
        b = (byte) i;
        b = (byte) 280; // = 24
        b = 128;       // *error* Java 1.1
    }
}
```

## C# : Primitive Type Casting

```
class PrimCast {
    public static void Main(string[] argv) {
        int i = 50;
        // *warning* The variable 'i' is assigned but its value is never used
        byte b = 127;
        b = (byte) 258;
        // b = 258;
        // *error* Constant value '258' cannot be converted to a 'byte'
        unchecked { b = (byte) 258; }
        System.Console.WriteLine("byte b = " + b);
        b = (byte) (b * 0);
        // b = (b * 0);
        // *error* Cannot implicitly convert type 'int' to 'byte'
    }
}
// Console Output: byte b = 2
```

## Boolean Type

- Distinct from integer type.
  - 0 (1) are *not* false (true). (Cf. C/C++)
- Let boolean *b, c* and int *i*.
  - `if (b) {} else {};`
  - `if (b == true) { c = false; }`  
`else { c = true; };`  
*equivalent to* `c = ! b;`
  - `if (i = 0) {};`  
is a type error (“`int used bool expected`”).

## Operators

- Variety
  - unary, binary, ternary
  - prefix, infix, postfix
  - arithmetic, bitwise, relational, logical
- `var op= expr; equivalent to var = var op expr;`
- Shift operators:
  - `i >> 5` (sign extension)
  - `i >>> 5` (unsigned)
  - `i << 5`
- Interaction with promotion (coercion) to `int`

## Boolean Operators

- Boolean logical *and/or* operator : `&, |`
- Boolean short-circuit *and/or* operator : `&&, ||`
  - `false && B == false`
  - `true || B == true`
- De Morgan’s laws
  - `!(a & b) = !a | !b`
  - `!(a | b) = !a & !b`
- Operator overloading :
  - `&` (resp. `|`) : bitwise/logical *and* (resp. *or*)
  - `+` : numeric add, string concatenation
    - *No user defined operator overloads.*

## Reference Types and Values

- **Class Types**
  - String is a class.
- **Interface Types**
- **Array Types**  
An *object* (resp. *array object*) is an instance of a class (resp. an array).  
A *reference* is a pointer to (or the address of) an object or an array object.

## Variables and Values

- A variable of a primitive type always holds a value of that *exact primitive type*.
- A variable of a reference type can hold either *a null reference* or *a reference to any object whose class is assignment compatible with the type of the variable*.

```
class Point { }
class ColoredPoint extends Point { }
class RefCast {
    public static void main (String [] args) {
        Object oR;
        Point pR = null;
        ColoredPoint cpR = null;
        oR = pR;
        pR = cpR;
        cpR = pR; // *error*
        cpR = (ColoredPoint) pR;
                // *run-time exception*

        X xR = null;
        cpR = (ColoredPoint) xR;
    }
}
class X { }
```

## Notes on Type Checking

- Coercion, Widening: `pR = cpR;`
- Casting, Narrowing:  
`cpR = (ColoredPoint) pR;`
  - Sound because a `Point`-type variable can potentially hold a `ColoredPoint` reference.
  - However, to guarantee type safety, Java compiler emits type checking code which can, at runtime, throw `ClassCastException` if the type constraint is not met.

## (cont'd)

- Casting, Narrowing:  
`xR = (X) pR;`
  - This is unsound because a `Point`-type variable can never hold an `X`-reference. So, compiler generates a type error.
- Initialization:
  - Java requires that local variables be explicitly initialized before their first use. So, assignments of null to `pR` and `cpR`, in the example, are mandatory.

## C# : Reference Type Casting

```
class Point { }
class ColoredPoint : Point { }
class X { }
class RefCast {
    public static void Main () {
        System.Object oR;
        Point pR = null;
        ColoredPoint cpR = null;
        oR = pR;
        pR = cpR;
        //cpR = pR;
        // *error* Cannot implicitly convert type 'Point' to 'ColoredPoint'
        cpR = (ColoredPoint) pR;
        X xR;
        // *warning* The variable 'xR' is declared but never used
        // xR = (X) pR;    // *error* Cannot convert type 'Point' to 'X'
    }
}
```

## Type Compatibility Examples

- class variable - subclass instance

```
import java.awt.*;          import java.applet.*;
Panel p = new Applet();
Applet a = (Applet) p;
p = a;
import java.io.*;
BufferedReader bin = new BufferedReader
    (new InputStreamReader (System.in));
```

- interface variable - class instance

```
Runnable p = new Thread();
```

## Subtle Differences

- Assignment ( $x = y$ )
  - Primitive type : copying a value
    - Parameter passing: *call by value*
  - Reference type: sharing an object
    - Parameter passing: *copying reference*
- *final* variable modifier
  - Primitive type : constant value
  - Reference type: constant object
    - Mutable : state of the object changeable
    - Immutable : state of the object constant
      - E.g., class String

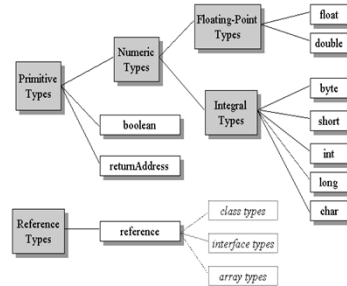
## Meaning of assignment

```
class ToAssign {
    public static void main(String[] argv) {
        int ip = 5;  Pair ir = new Pair (15,25);
        int jp = ip;  Pair jr = ir;

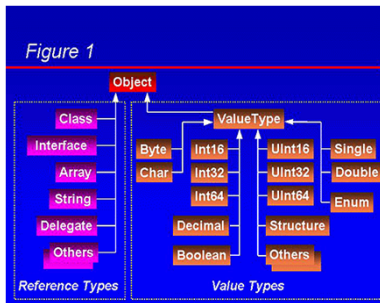
        System.out.println( "jp = " + jp + "\t" + "jr = " + jr);
        ip = 9;      ir.x = 19;
        System.out.println("ip = " + ip + "\t" + "ir = " + ir);
        System.out.println("jp = " + jp + "\t" + "jr = " + jr);
    }
}
```

- The uniform abstraction of “*everything is an object*” traded for efficiency.
  - No composite types such as *structures* or *unions*.
- Every expression has a type deducible at compile-time.
- All assignments of expressions to variables are checked for type compatibility.

### Data Types supported by JVM



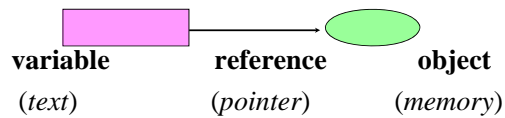
### Data Types supported by C#



### Motivation for Strong Typing

- Type declarations provide extra information associated with an identifier. This redundant info. enables mechanical detection of errors.
- In practice, a number of logical and typographical errors *manifest* themselves as type errors. Thus, strongly typed languages allow construction of reliable programs.
- In OOPs, the type tags associated with objects aid in the impl. of dynamic binding, polymorphism, and safe conversions.

## Typing



- **Static Typing** (e.g., Ada)
  - $type(variable) = type(object)$
  - compile-time checking : *efficient*
- **Dynamic Typing** (e.g., Scheme)
  - variables type-less; objects carry type tags
  - run-time type-checking : *flexible*

- **Typing in OOP** (E.g., Java, Eiffel, C++, ...)

- $type(object/reference)$  **is-a-subtype-of**  $type(variable)$ .
- In Java, a variable of class C, can hold a reference to an instance (object) of a subclass of C.
- Type correctness guaranteed at compile-time.
  - Efficient and secure.
- Dynamic binding dispatches each call to the appropriate code, at run-time.
  - Flexible handling of heterogeneous data.

## Arrays

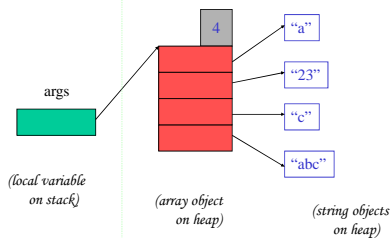
- **Declaration**
  - `int[] intArray; String[] args;`
- **Creation**
  - `intArray = new int[5];`
  - `intArray = { 0, 2, 4, 3*2, 4+4 };`
- **Array of arrays** (cf. multi-dimensional array)
  - `double [][] iA = { { 1, 0 }, null };`
- Java runtime verifies that array indices are in the correct range.

## An Example Program

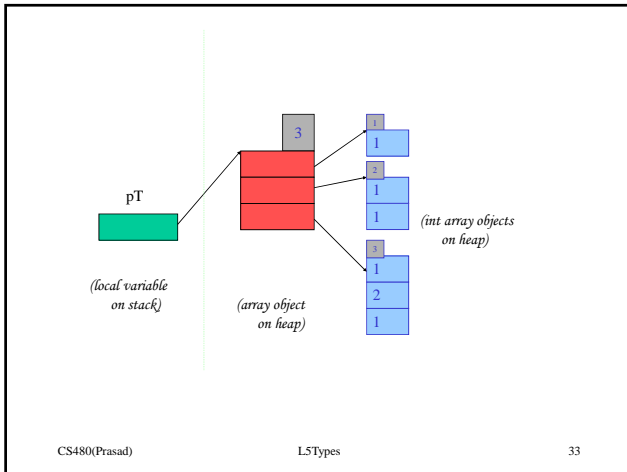
```
class EchoArgs {
public static void main(String[] args){
    for (int i = 0; i < args.length; i++)
        System.out.print( args[i] );
    };
    System.out.println( " ");
}
- javac EchoArgs.java
- java EchoArgs a 23 c abc
- java EchoArgs
```

## Java 5 version

```
class EchoArgs {
public static void main(String[] args){
    System.out.print("Command line arguments: ");
    for (String s : args)
        System.out.printf( " %s ", s );
    System.out.println(".");
}
}
- javac EchoArgs.java
- java EchoArgs a 23 c ab
```



```
class PascalTriangle {
public static void main( String[] args ) {
    n = Integer.parseInt( args[0] );
    int [] [] pT = new int [n + 1] [];
    pT[0] = new int[1]; // first row
    pT[0][0] = 1;
    for (int i = 1; i <= n; i++) { // rows 2 to n+1
        pT[i] = new int [i + 1];
        pT[i][0] = 1;
        pT[i][i] = 1;
        for (int j = 1; j < i ; j++ ) {
            pT[i][j] = pT[i-1][j-1] + pT[i-1][j];
        }
    }
}
}
```



### C# Equivalent

```

using System;
class PascalTriangleInCSharp {
    public static void Main( string[] args ) {
        int n = 7;
        if (args.Length > 0)
            n = int.Parse( args[0] );
        int [,] pT = new int [n + 1] [];
        pT[0] = new int[1]; pT[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            pT[i] = new int [i + 1];
            pT[i][0] = 1;
            pT[i][i] = 1;
            for (int j = 1; j < i ; j++) {
                pT[i][j] = pT[i-1][j-1] + pT[i-1][j];
            }
        }
    }
}

```

CS480(Prasad) L5Types 34

### C# Alternative

```

using System;

class RectArrayInCSharp {
    public static void Main( string[] args ) {
        const int n = 7;
        int [,] pT = new int [n,n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                pT[i,j] = i + j;
            }
        }
        System.Console.WriteLine(pT);
    }
}

```

CS480(Prasad) L5Types 35

- ### Array type : “unconstrained”
- Define matrix operations in terms of structure of the array (dimensionality); size is implicit.
    - *Vector dot-product* that works for two equal length single-dimension array with same type elements.
    - *Matrix multiplication* : Dimension compatibility checked at run-time.
  - Length field of an array object used :
    - to control loops.
    - to allocate storage for local variables.
  - Type is *unconstrained*, but each object/instance has a *fixed* size. (E.g., String type vs String object)
  - Cf. Ada Unconstrained array types.
- CS480(Prasad) L5Types 36

# Java Generics

```
import java.util.*;
public class OldList {
    public static void main(String args[]) {
        List list = new ArrayList();
        // (Potentially heterogeneous collection)
        list.add("one");
        list.add("two");
        list.add("three");
        list.add("four");
        Iterator itr = list.iterator();
        while(itr.hasNext()) {
            // Explicit type cast needed
            String str = (String) itr.next();
            System.out.println(str + " is " +
                str.length() + " chars long.");
        }
    }
}
```

```
import java.util.*;
public class NewList {
    public static void main(String[] args) {
        List <String> list =
            new LinkedList <String>();
        // Instantiating generic type.
        // (Homogeneous collection)
        list.add("one");
        list.add("two");
        list.add("three");
        list.add("four");
        for (String str : list) {
            System.out.printf("%s is %d chars long.\n",
                str, str.length());
        }
    }
}
```

```
using System;
using System.Collections.Generic;

public class NewList {
    public static void Main(string [] args) {
        // List class supports method Add
        LinkedList <String> list =
            new LinkedList <String>();
        // Instantiating generic type.
        // (Homogeneous collection)
        list.AddFirst("one");
        list.AddLast("two");
        list.AddLast("three");
        list.AddLast("four");
        foreach (string str in list) {
            System.Console.WriteLine(str + " is " +
                str.Length + " chars long.");
        }
    }
}
```