The Java Language

Language Elements

- Types
- Literals
- Variables
- Operators
- Control Structures
- Exceptions
- Arrays

First – some Java Philosophy

- Safety
  - avoid memory referencing errors, buffer overflow, ...
  - force the programmer to explicitly indicate that they know what they are doing!
    - leads to a *verbose* language – lots of code.
      - we should all now do some finger exercises to prepare!
    - the *plan* is to make it easier to write *correct* programs.
  - The compiler complains about of lot of things you might not expect.

The compiler will complain, can you guess where?

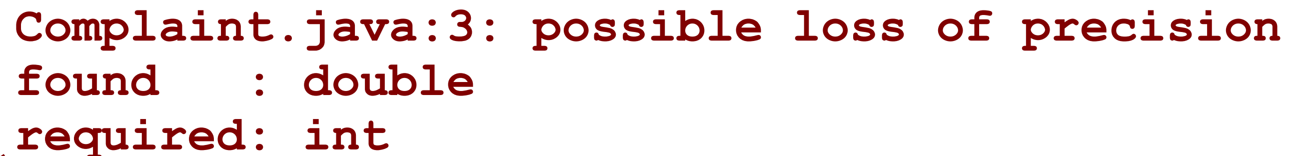
```
int i;
byte x;
float f;

i = 1.0;
x = 1;
x = x+1;
f=i;
```

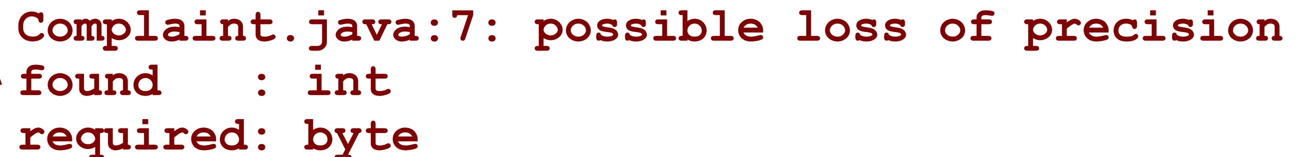
The compiler will complain, can you guess where?

```
int i;
byte x;
float f;

i = 1.0;
x = 1;
x = x+1;
f=i;
```



```
Complaint.java:3: possible loss of precision
found    : double
required: int
```



```
Complaint.java:7: possible loss of precision
found    : int
required: byte
```

A *clean* version (this will compile)

```
int i;
byte x;
float f;

i = (int)1.0;
x = 1;
x = (byte)(x+1);
f=i;
```

casting: telling the compiler to convert from one type to another.

cast 1.0 as an int.

cast x+1 as a byte.

Things to remember

- The compiler assumes you are a bad programmer.
- The compiler "always knows best".
- Don't argue with the compiler, it's a waste of time!
- When in doubt – cast something!
  - just kidding (sortof). You should never be in doubt!

Data Types

- Everything that has a *value* also has a *type*.
  - anything that can be stored in a variable, passed to a method, returned from a method or operated on by an operator.
- In Java there are two kinds of types:
  - Primitive Types
    - *Built-in data types.*
  - Reference Types
    - *Everything else, including objects*

Primitive Types

- **`char`**: unicode (2 bytes each!)

- Integral types
  - **`byte`**: 8 bit signed 2's complement integer
  - **`short`**: 16 bit signed 2's complement integer
  - **`int`**: 32 bit signed 2's complement integer
  - **`long`**: 64 bit signed 2's complement integer

Integer types and ranges

byte: `-128` to `127`

short: `-32,768` to `32,767`

int: `-2 billion` to `+2 billion` ← *roughly*

long: `-1 bazillion` to `+1 bazillion`.

Quiz: what data type should be used to represent a College President's annual income?

More Primitive Types

- Floating Point
  - **`float`** 32 bit IEEE 754
  - **`double`** 64 bit IEEE 754 (double precision)
- **`boolean`**: **`true`** or **`false`** only
  - not like C/C++: 0 doesn't mean false !
  - this may take some getting use to ...

Some General Rules

- Integer arithmetic is always done using 32 bit values (or more).
- You can't assign a 32 bit value to something that will only hold an 8 or 16 bit value
  - unless you cast it.
- Similar rules for floating point values
  - by default things are *double*, not *float*.

Computer Arithmetic

- In Java (and many other programming languages), arithmetic with integers and floating point values depend on fixed-size representations.
- The set of representable numbers is not closed under the arithmetic operations.
  - In other words, it's possible to have an operation for which the exact result cannot be represented!
- Computer arithmetic is not perfect!

Reference Types

- Everything that is not a primitive type is a *reference type*.
  - objects, arrays, interfaces
- A *reference* is a pointer, except that we are not allowed to call it a pointer since Java "doesn't have pointers".
  - OK – it's an address.
  - Actually, a pointer with very limited semantics, there isn't anything you can do with a reference except de-reference it.

Confused?

- Actually, Java is much simpler than C++!
  - I promise.
- Things will make more sense once we look at *variables*.
  - so you can forget everything I just said about references/pointers/addresses for now.

Literals

- Literals are fixed values found in a program.
  - we don't use the term “constant” for this, we use “constant” for something else...
- Examples:

```
x = y + 3;
System.out.println("Hello World");
finished = false;
```

Boolean Literals

- There are only three:

```
true
false
pancakes
```

Integer Literals

- Default is decimal (base 10). Examples

```
23        100        0        1234567
```

- Java also supports hexadecimal (base 16).

```
0x23      0x1      0xaf3c21      0xAF3C21
```

- and octal (base 8)

```
023      01      0123724
```

any integer literal that starts with a leading 0
is treated by Java as an octal number!

Long Literals

- Literal has the suffix '`l`' or '`L`'

```
23L   100l   0xABCDEF01234L
```

- Without the trailing '`l`' any integral literal is an `int` type (32 bit signed integer).

Floating Point Literals

- Decimal only (base 10).
- Optional suffix:
  - '**`f`**' or '**`F`**' means **`float`**
  - '**`d`**' or '**`D`**' means **`double`**

```
12.5    6.02E23D    1.5e-4
```

- The default is double, so you only need to use the 'F' suffix if you want something to be a float.

Character Literals

- characters are represented using unicode, not ASCII.
  - we (programmers) don't have to deal with this issue as much as you might think.
  - character literals look just like they do in C/C++

```
'a'    'b'    'C'    '\n'    '\t'
```

  - anything that starts with \u is treated as hex encoding of the unicode number

```
'\u03df'          '\u003f'
```

Other Literals

- There are also string literals

```
"Hello World"
```

OK, but what is a *string* in Java? Soon...

array literals (used only for initialization):

```
int a[] = {1,3,5,7,9};
```

Variables

- Storage location (chunk of memory) and an associated type.
  - type is either a primitive type or a reference type.
- For primitive type variables, a variable holds the actual value (the memory is used to store the value).
- For reference type variables, a variable holds a reference to an object (or array).

Variable Names

- Variable names are *identifiers* (so are class names, interface names, method names, ...)
- Identifiers are made of any length sequence of letters, digits and the underscore character '_'.
  - The character '$' is also supported, but is generally only used by code generators, not by humans.
- The first character of an identifier must not be a digit. (Otherwise it looks like a number!).

Kinds of variables

- There are lots of kinds of variables, we will bump in to the various kinds as we go...

class variables (static)

instance variables

parameters (method, constructor, exception handler)

local variables

array components

Operators

- arithmetic/logical
- assignment
- boolean/relational (comparison)
- string

Arithmetic/Logic Operators

- additive: `+` `-` *(unary and binary operators)*
- multiplicitive: `*` `/` `%`
- increment/decrement: `++` `--`
- bitwise: `&` `|` `^` `~`
- shift: `<<` `>>` `>>>`

Assignment Operators

- `=`
- `+=` `-=` `*=` `/=` `%=` `<<=` ...
  - You can use any of the arith/logic operators with assignment.

- An assignment expression has a value (just like in C/C++):

```
a = (b = 3);
x +=  (y -=2) +2;
```

Boolean/Relational Operators

- Equality: `==` `!=`
- Inequality: `<` `>` `<=` `>=`
- Conditional AND: `&&`
- Conditional OR: `||`

Comparison of reference types is an important issue (the above operators might not do what you think they do)!

The + String Operator

- Strings in Java are objects (just like any other object), except that the language includes special support:
  - String literals are part of the language.
  - The string concatenation operator **+** is part of the language.

```
System.out.println("Hello " + "World");
```

Control Structures

`if`

`if/else`

`do`

`while`

`for`

`switch`

Syntax is very similar (in most cases identical) to C/C++

if

```
if ( condition )
  statement;

if ( condition ) {
  statement1;
  statement2;
  …
}
```

example code

```
if ( a < 12 )
  b = 45;

if ( x.length()<10) {
  x = x + "BLAH";
  y=x.length()-3;
}
```

if else

```
if ( condition )
  statement;
else
  statement;
```

```
if ( a < 12 )
  b = 45;
else {
  if ( x<10 || y>2 )
    z=3;
}
```

while loop

```
while ( condition )
  statement;
```

```
i = 0;
x = 0;
while (i < 10) {
   x = x + i;
   i++;
}
```

do

```
do
  statement;
while ( condition )
```

```
i=0; x=0;
do {
  x += i;
  i++;
} while ( i<10 );
```

for

```
for ( initialization; condition; incrementor )
    statement;
```

initialization code is executed only once (before anything else)

condition is evaluated before ***statement*** is executed (every time).

incrementor code is executed after ***statement*** (every time).

`for` examples

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

for (int j=10; j>=0; j=j-2 ) {
  System.out.println("j is " + j);
   if (j>x)
      break;
}
```

New for loops

- Java 5 includes some new for loops that allow you to easily iterate over elements in an array or collection. For example:

```
for (String s : args) {
  System.out.println("arg is " + s);
}
```

switch

```
switch ( int expression ) {
   case int expression :
     statement;
     break;
   case int expression :
     statement;
     break;
   default :
     statement;
}
```

switch example

```
switch(food) {
  case 1:
    System.out.println("Chicken");
    break;
  case 2:
    System.out.println("Pizza");
    break;
  default:
    System.out.println("Sorry, we are out");
}
```

break and **continue**

- The **break** statement is similar to C++:
  - get out of a loop or case
- **break** can also specify a loop *target* (a label in the code). This allows **break** to jump out of nested loops.
- **continue** is used in loops to jump back to the beginning of the loop (skipping any statements between the **continue** and the end of the loop).

break examples

```
for (int i=0;i<10;i++) {
   System.out.println("i is " + i);
   if (i==3) break;
}

outer: for (int j=0;j<5;j++) {
   for (int k=0;k<5;k++) {
     if (k==3) break outer;
     System.out.println("j,k: " + j + "," + k);
   }
}
```

`continue` examples

```
for (int i=0;i<10;i++) {
   if (i==3) continue;
   System.out.println("i is " + i);
}

outer: for (int j=0;j<5;j++) {
   for (int k=0;k<5;k++) {
     if (k==3) continue outer;
     System.out.println("j,k: " + j + "," + k);
   }
}
```

Exceptions

- The Java language provides support for handling errors (or any kind of unusual condition).
- This is also available in C++, although it is not required (and often not used).
- If you want to use any Java libraries you need to understand exceptions (and you can't do much of anything in Java unless you use libraries!).

Exception Handling

- When an *exception* occurs, control is *thrown* to a special chunk of code that deals with the problem.
- This frees your chunks of code from needing lots of error handling code.
  - you still have to write code to deal with errors, but it goes in a special place. One chunk of error handing code can take the place of lots of error handling/checking code.

Advantages of Exceptions

- You don't need to return error codes from methods.
  - instead you indicate errors by using a special construct : you *throw* an error.
- You do need to think about what kind of errors can occur.
  - In some other languages you are free to ignore the possibility of errors. Is this a feature or weakness?

try/catch

```
try {

    statements . . .

} catch (ExceptionType ename) {

    error handling statements . . .

}
```

This is the code that might generate exceptions (error conditions)

This is the code that deals with exceptions.

`try/catch` examples

```
try {
   readFromFile("datafile");
} catch (FileNotFoundException e) {
   System.err.println("Error: File not found");
}
```

```
try {
   readFromFile("datafile");
} catch (Exception e) {
   System.err.println("Error: " + e );
}
```

Multiple **catches**

```
try {
    statements . . .
} catch (ExceptionType1 ename1) {
    error handling statements . . .
} catch (ExceptionType2 ename2) {
    error handling statements . . .
} catch (ExceptionType3 ename3) {
    error handling statements . . .
}
```

Generating Exceptions

- Many libraries (objects) generate exceptions when things go wrong. You will need to catch those exceptions!
- You can generate an exception with the **`throw`** keyword:

```
if (x > 100)
  throw new Exception("x is too big");
```

Exceptions are objects

- There are many exception types that are already defined (and documented in the Java API docs).
- Exceptions types are defined as an object type (a class).
  - we will look at this more closely once we deal with creating classes and inheritance.
  - If you define an exception named "up", consider what your code looks like when you throw this exception!

Arrays

- Arrays are objects.
  - Sort-of.
  - With special support in the language.
- Array syntax is very similar to C/C++
- Every array access is checked (at run time) to make sure it is within the bounds of the array.
  - arrays have fixed sizes (like in C/C++).

Array of what?

- You can have an array of anything, but all the elements of an array are the same type.
  - any primitive type.
  - any reference type.
- You need to declare the type of an array (the type of each element).
- You also need to explicitly create an array
  - just declaring the type is not enough.

Array variable declaration

```
element_type [] variablename;
```

Examples:

```
int [] foo;
String [] studentNames;
```

These variable now exist, but there are not yet any arrays!

`foo` and `studentNames` are reference types, they *can* refer to an array (but so far they don't).

Creating an Array

- You use the Java **new** operator:

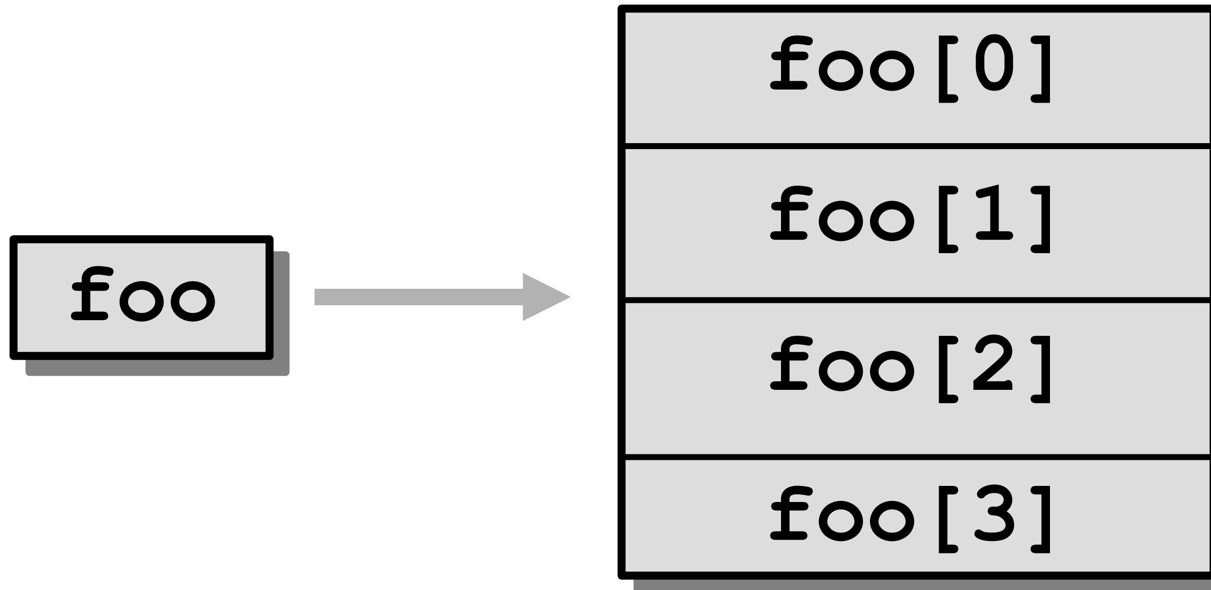
```
foo = new int [100];
studentNames = new String[20];
```

- You often see both declaration an creation like this:

```
int [] foo = new int [100];
String [] studentNames = new String[20];
```

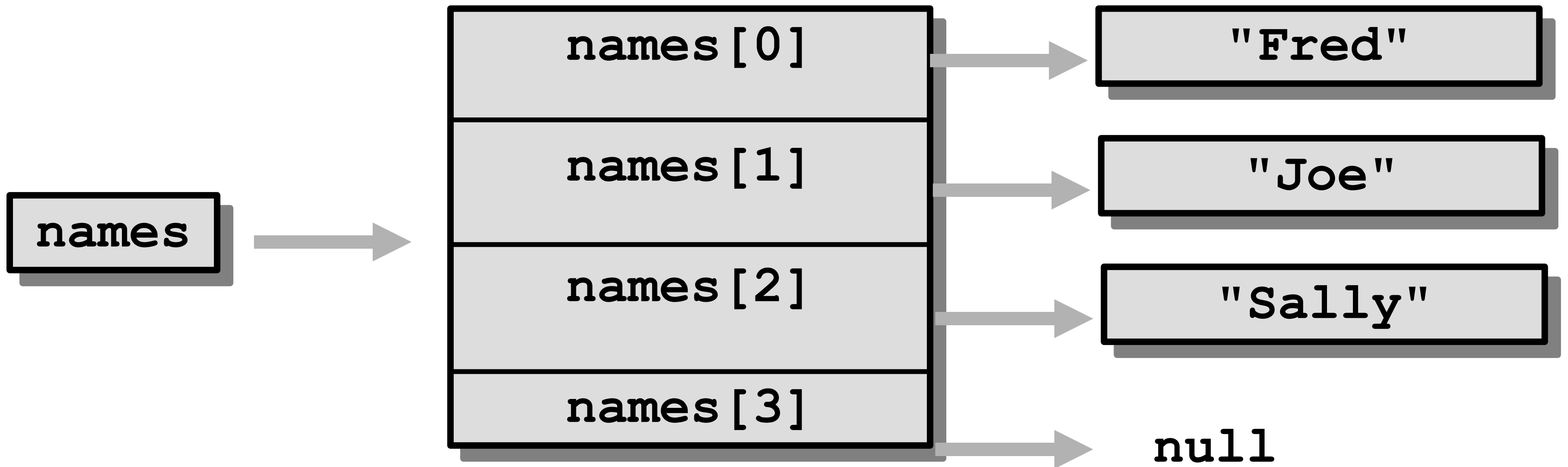
An important picture

```
int [] foo = new int [4];
```

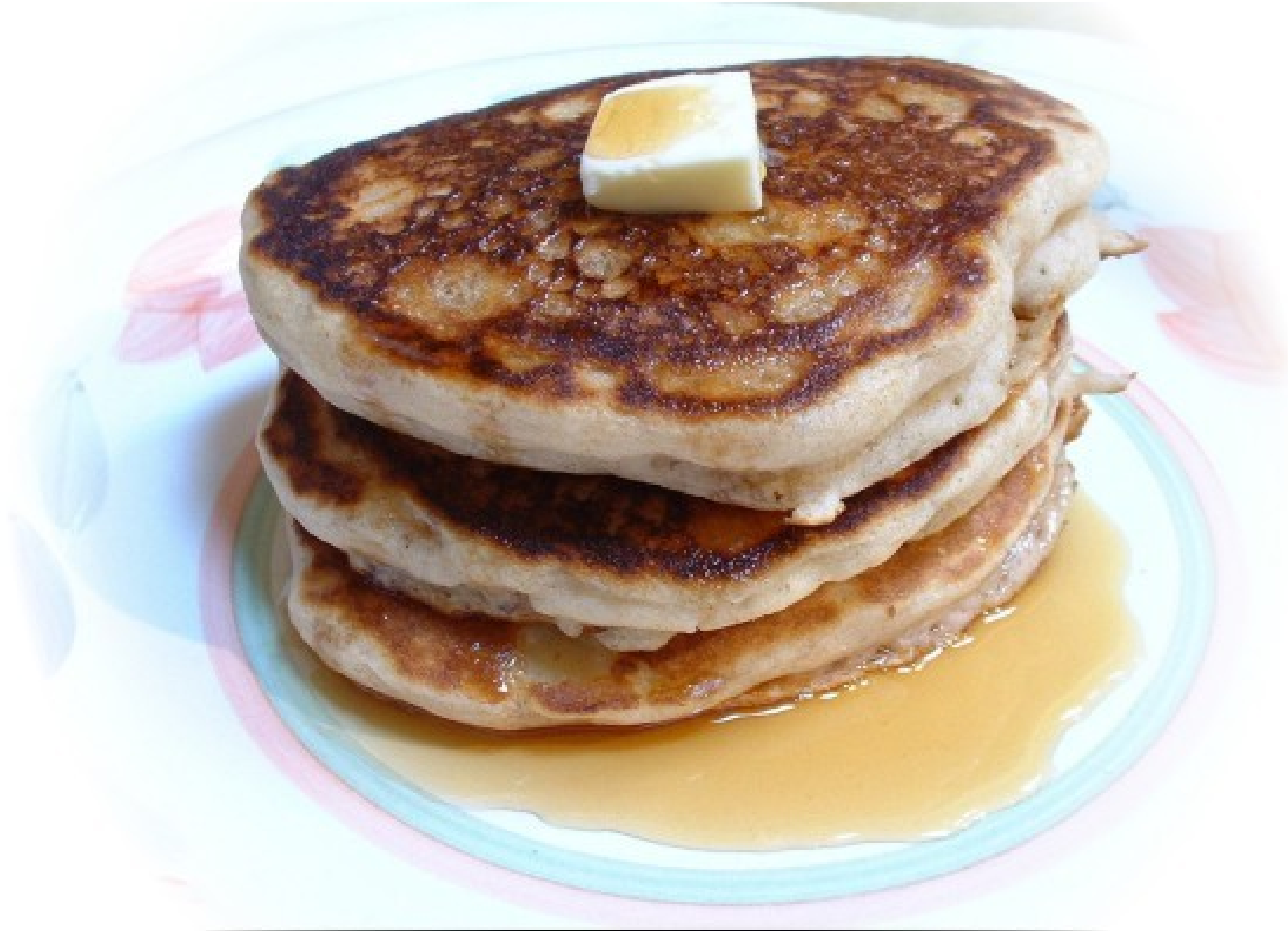


An even more important picture

```
String [] names = new String [4];
```



Possibly the most important picture?



Array Length

- The length of an array is available as a *field* of the array.

```
byte [] buff = new byte[100];
for (int i=0;i<buff.length;i++)
  buff[i]=i;
```

Array initializers

- You can use an *array literal* to create an array
  - you assign the literal to the array variable.

```
int [] foo = { 8, 2, 1, 4, 5 };
for (int i=0;i<foo.length;i++)
   System.out.println("foo["+i+"]="+foo[i]);
```