Type conversions

- Automatically applies to *promotions* only
  - e.g., `int n = 5; double d = n;` // okay
    - `n` is "promoted" to `double` before assignment happens
  - e.g., `int n = 5; double d = n/2.0;` // okay
    - `n` promoted to `double` before division; result is `double`
- Must "cast" to force other conversions
  - e.g., `double d = 5.; int n = d;` // error
    `double d = 5.; int n = (int)d;` // okay
  - But not all casts are legal (basically must make sense):
    `String s = "dog"; int n = (int)s;` // error

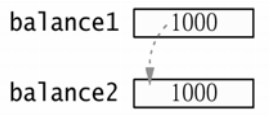
Some object reference issues

- `null` – a reference to no object at all
  - Cannot send message to "no object", of course
    - e.g., `BankAccount mySavings = null;`
      `mySavings.withdraw(100);` // error at runtime
- `this` – an object's reference to itself
  - Often used just for clarity:
    - e.g., in a BankAccount method, `balance = 0` is same as `this.balance = 0`
  - Also used to call one constructor from another one
    - e.g., `public BankAccount() { this(0); }`
- Copying a reference does *not* copy the object.

Copying values …

- Copying values (of primitive data types) actually does copy the value:

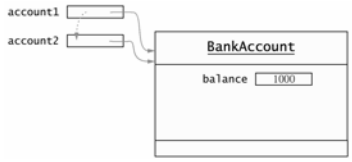
```
int balance1 = 1000;
int balance2 = balance1;
 // now there are separate copies of the value 1000
```



… vs. copying references

- Copying references to objects does *just* that:

```
BankAccount account1 = new BankAccount(1000);
BankAccount account2 = account1;
 // now there are separate copies of the reference
 // but there is still just one bank account object
```



Java has 7 control structures

- 1st is trivial: sequence structure
- 3 choices of selection structures (decisions):
  - `if`
  - `if/else`
  - `switch`
- 3 choices of iteration structures (loops):
  - `while`
  - `for` *(Two versions of it, since Java 5)*
  - `do/while`

`if`

Either

```
if (boolean expression)
    one statement;
```

Or

```
if (boolean expression) {
    multiple statements;
     separated by ;
}
```

Indented here too

Note indentation

- `boolean` expressions: evaluate to `true` or `false`

Simple `boolean` expressions

- Relational operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
  - e.g., `int x=1, y=2, z=3;`
    `x > y` // false
    `x >= z - y` // true
    - Note lower precedence than arithmetic
      `x == z + y` // false
    - Note not same as `x = z + y` // makes x be 5
      `z != x + y` // false (if x still is 1)

Boolean operators: `&&`, `||`, `!`

- For combining simple boolean expressions into more complex expressions
  - Operands are boolean expressions
  - e.g., `grade == 'A' && weight > 10`
    - Note: relational operators have higher precedence
- Truth tables: see text page 207
  - `op1 && op2` - true if *both* operands are true
  - `op1 || op2` - true if *either* operand is true
  - `!op` - true if operand is false
- Note: `&&` has greater precedence than `||`

`if/else`

1. Can implement with `if` and `else`:
```
if (grade >= 60)
    message = "Pass";
else
   message = "Fail";
```
2. Or with selection operator:
```
message = grade >= 60 ? "Pass" : "Fail";
    // same result as if/else above
```
   - This version does not allow {blocks} or nesting; but it returns a value, so more useful in many cases

`switch`

```
switch (controlling integral expression) {
   case constant integral expression:
        statements;
        break; // important
   case constant integral expression:
        statements;
        break;
   ...
   default:
        statements to do if no case matches;
}
```

`while`

```
while (boolean expression)
    operation; // or a block, delimited by { }
```

- Apply to loops for which termination uncertain
  - e.g., processing the "tokens" in a string
  - e.g., reading unlimited lines of input data
  - e.g., waitForBalance in Investment.java (p. 230)
- Awkward but usable for counter-controlled loops
```
int counter = 0;   // initialize
while (counter < 10) {   // compare to limit
    ...
    ++counter;   // increment (or otherwise change)
}
```

`for`

- More natural for counter-controlled loops:

initialize compare increment

```
for (int c = 0; c < 10; c++) ... // or {…}
```

  - e.g., waitYears in upgraded Investment.java (p. 239)
- Notes:
  - Header *requires* three fields (i.e., always two "`;`")
  - Watch scope of control variable
  - Alternate version for collections (upcoming)

do/while

```
do {
   . . . // loop body
} while (boolean expression);
```

- Notes:
  - Always executes at least once
    - e.g., good for user input checking
  - Don't forget the semicolon at the end

Arrays

- Definition: a *fixed* number of consecutive *memory* locations, all of the *same type*.
  - Either primitive data *values* of the same type
  - Or *references* to any one class of objects
- Can refer to all as a group by array's `name`
- Can refer to any one by `name[position]`
  - Position is called array "subscript" or "index"
  - First position is `0` (others are "offset" from 0)
- Also – in Java – an *object* in its own right

Arrays as objects

- A `public final` instance variable: `length`
  - Length is *fixed* after created (instantiated)
  - Range of positions: `0 ... length-1`
- Declare, instantiate – separate steps:

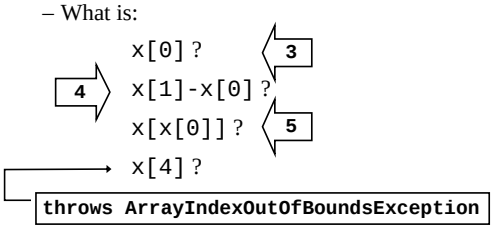
```
int x[]; // declare array of int named x
  • int[] x; // same thing (clear that x is an int array)
x = new int[4]; // instantiate array of length 4
      // all elements initialized to default values (0 in this case)
```

  - Or shortcut will instantiate and initialize elements

```
int x[] = { 3, 7, 4, 5 };
```

Accessing array elements

- Say `int x[] = { 3, 7, 4, 5 };`
  - What is:



Handling arrays

- `for` loops are especially useful:

```
for (int i=0; i < x.length; i++)
  { /* use x[i] in the loop body */ }
```

- Copy of reference is just an alias to same array

```
int[] a = x; // if x is an int array already
```

- Actual copy is a new object with copies of values

```
int[] a = new int[x.length]; // same length as x
for (int i=0; i < x.length; i++)
      a[i] = x[i];
```

Enhanced **`for`** loop (since Java 5)

- Actually a "for each" loop

```
for (int element : array)
```

  - Reads "for each element in array"
- e.g., array of strings: `String words[] = …`

```
for (String s : words)
    System.out.println(s);
```

- Note the loop control variable is the array element itself, not its array index
  - So not applicable if index value is required

Aside: `enum` demo

Basic array techniques

- Summing array elements:

```
int sum = 0;
for (int item : x)
    sum += item;
```

- Finding a maximum (or other extreme):

```
int max = x[0]; // initialize to first value
for (int i=1; i < x.length; i++)
    if (x[i] > max) max = x[i];
```

- Printing on one row of standard output:

```
for (int item : x) System.out.print(" " + item);
System.out.println(); // newline after row is done
```

  - Q: How to print in reverse order?