CSCI 136
Data Structures & Advanced Programming

Spring 2021

Instructors

Sam McCauley & Bill Lenhart

Java Over/Review (Crash Course)

Part II:
Primitive Types
Array Types
Operators and Expressions
Control Structures

Primitive Types

- Provide numeric, character, and logical values
  - 11, -23, 4.21, ‘c’, false
- Can be associated with a name (*variable*)
- Variables *must* be declared before use

```
int age;      // A simple integer value
float speed; // A number with a ‘decimal’ part
char grade;  // A single character
bool loggedIn; // Either true or false
```

- Variables *can* be initialized when declared

```
int age = 21;
float speed = 47.25;
char grade = ‘A’;
bool loggedIn = true;
```

Primitive Types

- Uninitialized *instance variables* (variables that describe features of an object) of primitive type are given default values

```
int age;     // Initialized to 0
float speed; // Initialized to 0.0
char grade; // Initialized to \u0000 (Unicode)
bool loggedIn; // Initialized to false
```

- Uninitialized local variables declared in a method are *not* given default values
  - Rule of Thumb: Always initialize a local variable when you declare it!

Strings and Primitive Types

Often numeric data is made available as a string

- Consider a class Sum which adds two numbers provided on the command line
  - java Sum 3 5, for example, would return 8
- 3 and 5 are held as String values in `args[]`
- The values can be converted to `int` values using the `valueOf()` method of the class `Integer`

```
int num1 = Integer.valueOf( args[0] );
```

- Similar classes exist for other primitive types

```
double pi = Double.valueOf("3.14159");
boolean notDone = Boolean.valueOf("true");
```

- However (gotcha!):

```
char initial = "William".charAt(0);
```

Sum

```
public class Sum {

   public static void main(String[] args) {

      if ( args.length < 2 )
         System.out.println( "Syntax: java Sum3 num1 num2" );

      else {
         int n0 = Integer.valueOf( args[0] );
         int n1 = Integer.valueOf( args[1] );
         System.out.println(n0 + " + " + n1 + " = " + (n0 + n1));
      }
   }
}
```

Array Types

- Holds a collection of values of some type
- Can be of any type

```
int[] ages;        // An array of integeras
float[] speeds;    // An array of floats
char[] grades;     // An array of characters
bool[] loggedIn;   // Either true or false
```

- Arrays can be initialized when declared

```
int[] ages = { 21, 20, 19, 19, 20 };
float[] speeds = { 47.25, 3.4, -2.13, 0.0 };
char[] grades = { 'A', 'B', 'C', 'D'  };
bool[] loggedIn = { true, true, false, true };
```

- Or just created with a standard default value

```
int[] ages = new int[15]; // array of 15 0s
```

Array Types

Notes

- Arrays are not primitive types in Java, they are class types (an array is therefore an *object* in Java)
- As a result, an uninitialized array holds the special object value `null`. This means
  - It is an error to attempt to index into the array

    ```
    int[] scores;      // Uninitialized array
    scores[0] = 100;   // Error!
    ```

  - It is an error to access any instance variable or method of an uninitialized array

    ```
    int[] scores;      // Uninitialized array
    int size = scores.length;     // Error!
    ```

Example: Rolling a Die

```
import java.util.Random;      // importing an external class

public class DieRoller {

  public static void main(String[] args) {
    // A random number generator
    Random rng = new Random();
    int faces = Integer.valueOf(args[0]);
    int[] counts = new int[faces];  // initialized to 0s
    int numRolls = 100*faces;       // number of tests

    // generate numRolls random values in range 0..faces-1
    for (int i = 0; i < numRolls; i++)
      counts[rng.nextInt(faces)]++;

    for (int i = 0; i < faces; i++)
      System.out.println(""+ i + ": " + counts[i]);
  }
}
```

Operators

Java provides a number of *operators* including

- Arithmetic operators: +, -, *, /, %
- Relational operators: ==, !=, <, ≤, >, ≥
- Logical operators &&, || (don't use &, |)
- Assignment operators =, +=, -=, *=, /=, ...

Common unary operators include

- Arithmetic: - (prefix); ++, -- (prefix and postfix)
- Logical: ! (not)

Operator Gotchas!

- There is no exponentiation operator in Java.
  - The symbol ^ is the *bitwise xor* operator in Java.
- The *remainder* operator % is the same as the mathematical 'mod' function for *positive* arguments,
  - For **negative** arguments **it is not**: -8 % 3 = -2
- The logical operators && and || use *short-circuit evaluation*:
  - Once the value of the logical expression can be determined, no further evaluation takes place.
  - E.g.: If n is 0, then `(n != 0) && (k/n > 3)`, will yield false without evaluating k/n. Very useful!

Example

```
/* QuotientRemainder.java
 * Prints the quotient and remainder given positive integers
 * java QuotientRemainder 23 4 returns Q = 5, R = 3
 */
public class QuotientRemainder {

  public static void main(String[] args) {
    if(args.length != 2 )
      System.out.println("Usage: java QuotientRemainder int
int ");
    else {
       int a = Integer.valueOf(args[0]);
       int b = Integer.valueOf(args[1]);
       System.out.println("Q = " + a/b + ", R = " + a % b);
    }
  }
}
```

Expressions

Expressions are either:

- literals, variables, invocations of non-void methods, or
- statements formed by applying operators to them

An expression returns a value

- `3+2*5 - 7/4  // returns 12`
- `x + y*z – q/w`
- `(- b + Math.sqrt(b*b – 4 * a * c) )/( 2* a)`
- `( n > 0) && (k / n > 2) // computes a` *`Boolean`*

Operator Precedence

- 3*5+2 = 2+3*5

Operator Precedence in Java

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Expressions

Assignment operator also forms an expression

- x = 3; // assigns x the value 3 and returns 3
- So y = 4 * (x = 3) sets x = 3 and y = 12 (and returns 12)

Boolean expressions let us control program *flow of execution* when combined with *control structures*

Example

- `if ( (x < 5) && (y !=0 ) ) {...}`
- `while (! loggedIn) { ... }`

Control Structures

Select next statement to execute based on value of a boolean expression. Two flavors:

- Looping structures: `while, do/while, for`
  - Repeatedly execute same statement (block)
- Branching structures: `if, if/else, switch`
  - Select one of several possible statements (blocks)
  - Special: break/continue: exit a looping structure
    - break: exits loop completely
    - continue: proceeds to next iteration of loop

while & do-while

Consider this code to flip a coin until heads comes up...

```
Random rng = new Random();
int flip = rng.nextInt(2), count = 1;
while (flip == 0) {   // count flips until “heads”
    flip = rng.nextInt(2);
    count++;
}
```

...and compare it to this

```
int flip, count = 0;
do {                  // count flips until “heads”
    flip = rng.nextInt(2);
    count++;
} while (flip == 0) ;
```

For & for-each

Here's a typical **for** loop example

```
int[] grades = { 100, 78, 92, 87, 89, 90 };
int sum = 0;
for( int i = 0; i < grades.length; i++ )
    sum += grades[i];
```

This **for** construct is equivalent to

```
int i = 0;
while ( i < grades.length ) {
    sum += grades[i];
    i++;
}
```

Can also write

```
for (int g : grades ) sum += g;
// called for-each construct
```

Loop Construct Notes

- The body of a **while** loop may not ever be executed
- The body of a **do – while** loop always executes at least once
- **For** loops are typically used when number of iterations desired is known in advance. E.g.
  - Execute loop exactly 100 times
  - Execute loop for each element of an array
- The **for-each** construct is often used to access array (and other collection type) values when *no updating* of the array is required
  - We'll explore this construct more later in the course

If/else

```
if (x > 0)          // There is exactly 1 "if" clause
      y = 1 / x;
else if (x<0) {     // 0 or more "else if" clauses
      x = - x;
      y = 1 / x;
}
else                // at most 1 "else" clause
      System.out.println("Can't divide by 0!");
```

The single statement can be replaced by a *block:* any sequence of statements enclosed in {}

switch

Example: Encode clubs, diamonds, hearts, spades as 0, 1, 2, 3

```
int x = myCard.getSuit(); // a fictional method
switch (x) {
   case 1: case 2:
      System.out.println("Your card is red");
      break;
   case 0: case 3:
      System.out.println("Your card is black");
      break;
   default:
      System.out.println("Illegal suit code!");
      break;
}
```

Break & Continue

Suppose we have a method `isPrime` to test primality

Find first prime > 100

```
for( int i = 101; ; i++ )
    if ( isPrime(i) ) {
        System.out.println( i );
        break;
    }
```

Print primes < 100

```
for( int i = 1; i < 100 ; i++ ) {
    if ( !isPrime(i) )
        continue;
    System.out.println( i );
}
```

Summary

Basic Java elements so far

- Primitive and array types
- Variable declaration and assignment
- Operators and expressions
- Java control structures
  - for, for-each, while, do-while
  - Switch, break, continue