Java Basics 1

Lecture 1

By Marina Barsky

Primitive Types
Variables
Array Types
Operators and Expressions
Control Structures

Primitive Types

- Provide numeric, character, and logical values

  11, -23, 4.21, ‘c’, false

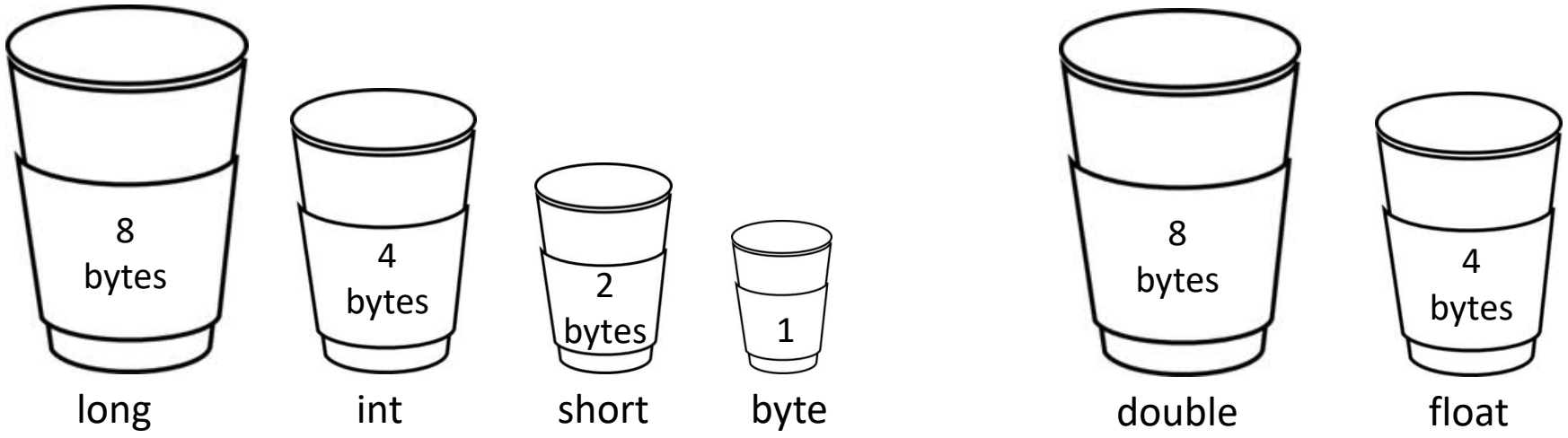
- Can be associated with a name (*variables*)
- Variables *must* be declared before use

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

- Variables *can* be initialized when declared

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

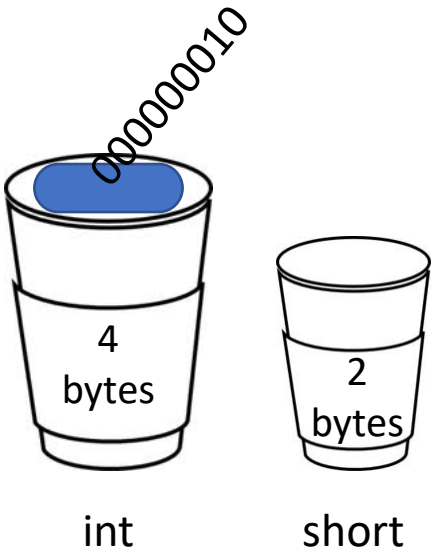
Size of numeric variables



Casting from one type to another

int t=2;

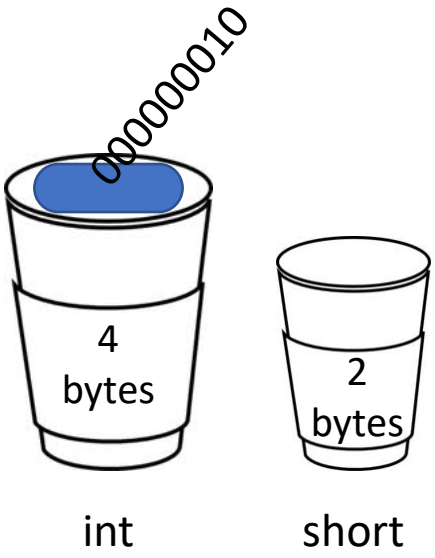
short s = t;



Casting from one type to another

int t=2;
short s = t;

This will not compile!



int t=2;
*short s = **(short)** t;*

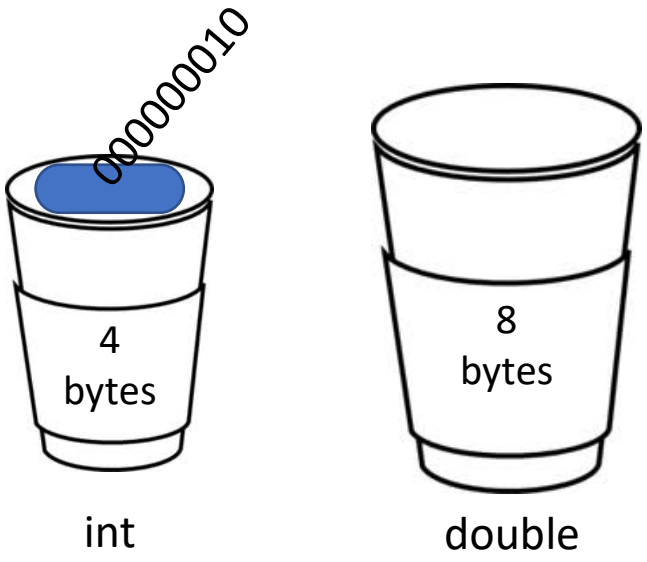
You cannot stick the value of a larger type into a smaller cup.

Compiler prevents loss of data.
It requires an **explicit casting**

No casting required

int t=2;
double d = t;

This is valid: can store an int in a double without loss of data



What is the value of i after:

```
double tau = 2 * 3.14;
int i = (int) tau;
```

A. i = 6.28
B. i = 6
C. i= 7
D. This will cause a compilation error
E. Something else

From Strings to 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, should 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);
```

Array Types

- Holds a collection of values
- Values can be of any type

```
int[] ages;         // An array of integers
float[] speeds;     // An array of floats
char[] grades;      // An array of characters
boolean[] 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'  };
boolean[] loggedIn = {true, true, false, true};
```

- Or just created with a standard default value

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

Array is always an object!

- Arrays are not primitive types in Java, they are reference 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]);
  }
}
```

Tests the quality of the random number generator, by counting the number of die sides and storing values in an array

Example: Sum 2

Sums up all the command-line arguments

```
public class Sum2 {

    public static void main(String[] args) {

        if ( args.length == 0 )
            System.out.println( 0 );

        else {
            int total = 0;
            for ( int i = 0; i < args.length; i++ )
                total = total + Integer.valueOf( args[i] );

            System.out.println( "The sum equals " + total );
        }
    }
}
```

What happens if one of the arguments is not int?

Example: Sum 3

```
public class Sum3 {
    public static void main(String[] args) {

        if ( args.length == 0 )
            System.out.println( 0 );

        else {
            int total = 0;

            // 'for-each' version of for loop
            for ( String num : args )
                total = total + Integer.valueOf( num );

            System.out.println( "The sum equals " + total );
        }
    }
}
```

Same as Sum 2: 'for-each' version of the FOR loop

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
- The result of the **division depends on the types of operands**:

```
8/3 = 2                          // both integers
8/3.0 = 2.6666666666666665    // one float
```

- 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!

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) // returns 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	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Control Structures

Select a different program path based on value of a boolean expression.

Two flavors:

- Looping structures: **`while`**, **`do/while`**, **`for`**
  - Repeatedly execute same statement (block of code)
- 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

```
int[] grades = { 100, 78, 92, 87, 89, 90 };
int sum = 0;
```

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

```
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 (read-only loop)

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 of code:* any number of statements enclosed in {}

Indentation does not matter in Java!

What is the value of x after:

```
int x = 0;
if (x == 5)
   x = 5;
   x+= 3;
System.out.println(x);
```

A. 0

B. 3

C. 8

D. -5

E. None of the above

Conditional Statements

Indentation does not matter in Java!

```
int x = 0;
if (x == 5) {
 x = 5;
 x+= 3;
}
System.out.println(x);
```

However, if you indent poorly, people who read your code will be unhappy with you.

switch

Prints message depending on student's grade:

```
char x = student.getGrade(); // a fictional method
switch (x) {
   case 'A':
      System.out.println("Good job!");
      break;
   case 'B': case 'C':
      System.out.println("Well done");
      break;
   default:
      System.out.println("Sorry - you failed");
}
```

switch

Prints message depending on student's grade:

```
char x = student.getGrade(); // a fictional method
switch (x) {
    case 'A':
        System.out.println("Good job!");
        break;
    case 'B': case 'C':
        System.out.println("Well done");
        break;
    default:
        System.out.println("Sorry - you failed");
}
```

switch

What is printed if grade is 'A'?

```
char x = student.getGrade(); // a fictional method
switch (x) {
   case 'A':
      System.out.println("Good job!");
      break;
   case 'B': case 'C':
      System.out.println("Well done");
      break;
   default:
      System.out.println("Sorry - you failed");
}
```

switch

The grade is still 'A'. What is printed **now**?

```
char x = student.getGrade(); // a fictional method
switch (x) {
   case 'A':
      System.out.println("Good job!");
   case 'B': case 'C':
      System.out.println("Well done");
   default:
      System.out.println("Sorry - you failed");
}
```

switch

What is printed **now**?

```
char x = student.getGrade(); // a fictional method
switch (x) {
    case 'A':
        System.out.println("Good job!");
    case 'B': case 'C':
        System.out.println("Well done");
    default:
        System.out.println("Sorry - you failed");
}
```

Without break – once entered – falls through all the remaining cases

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 );
}
```

This will print:

```
for (int i = 0; i < 10; i++) {
    System.out.print(i + "a");
    if (i == 0) {
        continue;
    }
    System.out.print("b");
    if (i == 1) {
        break;
    }
    System.out.println("c");
}
```

A. 0a
B. 0a1ab
C. 0ac1ab
D. 0ac1abc
E. None of the above

continue – start the next iteration of a loop

break – stop iterating a loop entirely

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
  - If-else, switch, break, continue

To do list

- ❑ Go over slides of Lecture 1
- ❑ Read demo code
- ❑ Listen to videos
- ❑ Do Home quiz 1
- ❑ Get iClicker and register it with blackboard

Announcement: iClickers

- We will be using iClickers for class participation.
- You can get your own clicker in the library
- After that you should go to blackboard and register your own device
- Alternatively, you can use mobile app instead of a physical device. The app is called iClicker Student, and is available at the App store of your mobile device.