CSCI 136
Data Structures &
Advanced Programming

Lecture 3
Fall 2017
Instructors: Bill & Bill
Administrative Details

• Lab today in TCL 216 (217a is available, too)
  • Lab is due by 11pm Sunday
    • Copy your folder to Dropoff folder for your lab (see handout)
• Lab 1 design doc is “due” at beginning of lab
  • Written design docs will be required at all labs
  • You’ll discuss with another student at start of lab
  • Several implementation options
    • Some may be better than others....
CoinStrip Design

• How to store game state?
  • Space needs
  • Time to find coin

• Useful methods?
  • void makeMove(whichCoin, howFar)
  • boolean legalMove(whichCoin, howFar)
  • toString()

• What, if anything, did lab description omit?
  • Form of “game board” to show players
Last Time

• Arrays, Operators, Expressions
• Some Simple Examples (Sum0-5)
  • Entering, editing, compiling, running programs
Today’s Outline

- Control structures
  - Branching: if – else, switch, break, continue
  - Looping: while, do – while, for, for – each
- Object oriented programming Basics (OOP)
- Strings and String methods
- More on Class Types
  - Interface specification for behavior abstraction
  - Inheritance (class extension) for code reuse
  - Abstract Classes
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 coin until heads up...

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

...and compare it to this

```java
int flip, count = 0;
do { // count flips until “heads”
    count++;
    flip = rng.nextInt(2);
} while (flip == 0) ;
```
Here’s a typical `for` loop example

```java
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

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

Can also write

```java
for ( int g : grades ) // called `for-each` construct
    sum += g;
```
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) // 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 {}
Example: Encode clubs, diamonds, hearts, spades as 0, 1, 2, 3

```java
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

```java
for(int i = 101; ; i++) // What's with ; ; ?
    if ( isPrime(i) ) {
        System.out.println( i );
        break;
    }
```

Print primes < 100

```java
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 & operator precedence
- Expressions
- Control structures
  - Branching: if – else, switch, break, continue
  - Looping: while, do – while, for, for – each
- Edit (emacs), compile (javac), run (java) cycle
Object-Oriented Programming

• Objects are building blocks of Java software

• Programs are collections of objects
  • Cooperate to complete tasks
  • Represent “state” of the program
  • Communicate by sending messages to each other
    • Through method invocation
Object-Oriented Programming

• Objects can model:
  • Physical items - Dice, board, dictionary
  • Concepts - Date, time, words, relationships
  • Processing - Sort, search, simulate

• Objects contain:
  • State (instance variables)
    • Attributes, relationships to other objects, components
      – Letter value, grid of letters, number of words
  • Functionality (methods)
    • Accessor and mutator methods
      – addWord, lookupWord, removeWord
Object Support in Java

• Java supports the creation of programmer-defined types called class types

• A class declaration defines data components and functionality of a type of object
  • Data components: instance variable (field) declarations
  • Functionality: method declarations
  • Constructor(s): special method(s) describing the steps needed to create an object (instance) of this class type
A Simple Class

Premise: Define a type that stores information about a student: name, age, and a single grade. Declare a Java class called Student with data components (fields/instance variables)

    String name;
    int age;
    char grade;

And methods for accessing/modifying fields

- **Getters**: getName, getAge, getGrade
- **Setters**: setAge, setGrade

Declare a constructor, also called Student
public class Student {
    // instance variables
    private int age;
    private String name;
    private char grade;

    // A constructor
    public Student(int theAge, String theName, char theGrade) {
        age = theAge;
        name = theName;
        grade = theGrade;
    }

    // Methods for accessing/modifying objects
// ...see next slide...
public int getAge() {return age;}

public String getName() {return name;}

public char getGrade() {return grade;}

public void setAge(int theAge) {
    age = theAge;
}

public void setGrade(char theGrade) {
    grade = theGrade;
}
} // end of class declaration
public class TestStudent {

    public static void main(String[] args) {
        Student a = new Student(18, "Bill J", 'A');
        Student b = new Student(21, "Bill L", 'A+');
        // Nice printing
        System.out.println(a.getName() + ", ", " +
            a.getAge() + ", ", " + a.getGrade());
        System.out.println(b.getName() + ", ", " +
            b.getAge() + ", ", " + b.getGrade());
        // Tacky printing
        System.out.println(a);
        System.out.println(b);
    }
}
Worth Noting

• We can create as many student objects as we need, including arrays of Students

```java
Student[] class = new Student[3];
class[0] = new Student(18, "Huey", 'A');
class[1] = new Student(20, "Dewey", 'B');
class[2] = new Student(20, "Louie", 'A');
```

• Fields are `private`: only accessible in Student class

• Methods are `public`: accessible to other classes

• Some methods return values, others do not
  - public `String` `getName()`;
  - public `void` `setAge(int theAge);`
A Programming Principle

Use constructors to initialize the state of an object, nothing more.

- State: instance variables
- Frequently they are short, simple methods
- More complex constructors will typically use helper methods.
- You constructors can call other constructors to reuse code
Access Modifiers

• **public** and **private** are called *access modifiers*
  • They control access of other classes to instance variables and methods of a given class
  • **public**: Accessible to all other classes
  • **private**: Accessible only to the class declaring it

• **There are two other levels of access that we’ll see later**

• **Data-Hiding (encapsulation) Principle**
  • Make instance variables **private**
  • Use **public** methods to access/modify object data
public class Student {
    // instance variables
    private int age;
    private String name;
    private char grade;

    // A constructor
    public Student(int age, String name, char grade) {
        // What would age, name, grade
        // refer to here...?
    }
}
public class Student {

    // instance variables
    private int age;
    private String name;
    private char grade;

    // A constructor
    public Student(int age, String name, char grade) {
        this.age = age;
        this.name = name;
        this.grade = grade;
    }
}
String in Java Is a Class Type

• Java provides language support for Strings
  • String literals: “Bob was here!”, “-113”, “A”, “”

• If a class provides a method with signature
  public String toString()
  Java will automatically use that method to produce a
  String representation of an object of that class type.

• For example
  System.out.println(aStudent);
  would use the toString method of Student to
  produce a String to pass to the println method.

Pro Tip: Always provide a toString method! It helps to
debug if you can visualize the state of your objects!
String methods in Java

- Useful methods (also check String javadoc page)
  - `indexOf(string) : int`
  - `indexOf(string, startIndex) : int`
  - `substring(fromPos, toPos) : String`
  - `substring(fromPos) : String`
  - `charAt(int index) : char`
  - `equals(other) : bool` ← Always use this!
  - `toLowerCase() : String`
  - `toUpperCase() : String`
  - `compareTo(string) : bool`
  - `length() : int`
  - `startsWith(string) : bool`

- Understand special cases!