

CSSE 220 Day 6

Console Input, Text Formatting,
Decision Statements and Expressions

Check out *Decisions* from SVN

Questions?

Grader Comments in Eclipse

- ▶ **How to see them (next slide = What to do with them)**
 1. Update your homework project: HW1 in this case
 - Right-click the project and select **Team** ⇒ **Update to HEAD**
 2. Examine your Tasks view
 - One of the tabs at the bottom of Eclipse
 - Use **Window** ⇒ **Show View** ⇒ **Other** ⇒ **General** ⇒ **Tasks** if needed
 - Your Tasks view has been configured to show all comments with **TOD**, **FIXME** and **CONSIDER** in them.
 - If you want to use other tags too, it's easy: Look at **Window** ⇒ **Preferences** ⇒ **Java** ⇒ **Compiler** ⇒ **Task Tags**
 3. Each **CONSIDER** “task” is a place where the grader has suggested an improvement to your code
 - The grader should make a **CONSIDER** for *every* place where the grader deducted points
 - Each homework has a link to its grading rubric.
 - Note especially the link in the grading rubric to **General Instructions for Grading Programs**

Grader Comments in Eclipse

▶ What to do with them: Earn Back!

- Within 3 days of receiving your project back, at each **CONSIDER**:
 1. Correct the error.
 2. Change the word **CONSIDER** to **REGRADE**
- The grader will re-grade any (but only) such tags. *If you correct all your errors, you earn back all the points that were deducted!*
- Some assignments will allow Earn Back, some won't. Earn Back *is* available for HW1.
- Earn Back is a privilege – don't abuse it. Put forth your “good faith” effort on the project and reserve Earn Back for errors that you did not anticipate.
- If the comment from the grader does not make clear what your error is:
 - First look at the grading rubric for the homework (and the link therein to General Instructions for Grading Programs).
 - Then ask questions as needed.

Grader Comments in Eclipse

▶ Some common errors from HW 1:

- *Leaving behind a TODO (either not doing the TODO or doing it but not erasing the TODO comment itself)*
- *Leaving behind compiler warning messages*
- Failing to put your own name as author of your classes
- *Using variable names that are not self-documenting*
- Not using the required names for the SeriesSum class and its method
- *Various formatting errors that Control-Shift-F corrects*
- Declaring a for-loop variable *outside* of the for-loop
- Using *double* as the return type for *factorial* or *seriesSum*
 - In general, use *int* or *long* for exact arithmetic. Using *double* opens the door for roundoff error.
- Not an error, just a comment: good practice to precede static fields with the class name, e.g. **Factorial.MAX** not just **MAX**

Outline

- ▶ String Input and Output
 - ▶ Quick review of **if** statements
 - ▶ **==** vs. **equals()**
 - ▶ Selection operator, **? :**

 - ▶ Optional: **switch** and enumerations
- 

char Type in Java is Like C's

- ▶ In Python:
 - “This is a string”
 - ‘and so is this’
- ▶ In Java:
 - “This is a string”
 - This is a character: ‘R’
 - ‘This is an error’

Iterating Over Strings in Java

- ▶ Can use **charAt(index)**

- ▶ Example:

```
String message = "Rose-Hulman";  
for (int i=0; i < message.length(); i++) {  
    System.out.println(message.charAt(i));  
}
```

- ▶ **charAt()** returns a 16-bit **char** value*
- ▶ Exercise: Work on TODO items in **StringsAndChars.java**

* Unfortunately there are more than 2^{16} (65536) symbols in the known written languages. See Character API docs for the sordid details.

Reading Console Input with `java.util.Scanner`

- ▶ Creating a Scanner object:
 - `Scanner inputScanner = new Scanner(System.in);`
- ▶ Defines methods to read from keyboard:
 - `inputScanner.nextInt()`
 - `inputScanner.nextDouble()`
 - `inputScanner.nextLine()`
 - `inputScanner.next()`
- ▶ Exercise: Look at [ScannerExample.java](#)
 - Add `println`'s to the code to prompt the user for the values to be entered

Formatting with `printf` and `format`

Table 3 Format Types

Code	Type	Example
d	Decimal integer	123
x	Hexadecimal integer	7B
o	Octal integer	173
f	Fixed floating-point	12.30
e	Exponential floating-point	1.23e+1
g	General floating-point (exponential notation used for very large or very small values)	12.3
s	String	Tax:
n	Platform-independent line end	

Table 4 Format Flags

Flag	Meaning	Example
-	Left alignment	1.23 followed by spaces
0	Show leading zeroes	001.23
+	Show a plus sign for positive numbers	+1.23
(Enclose negative numbers in parentheses	(1.23)
,	Show decimal separators	12,300
^	Convert letters to uppercase	1.23E+1

More options than in C.
I used a couple in
today's examples.
Can you find them?

Formatting with `printf` and `format`

▶ Printing:

- `System.out.printf("%5.2f%n", Math.PI);`

▶ Formatting strings:

- `String message =
String.format("%5.2f%n", Math.PI);`

▶ Display dialog box messages

- `JOptionPane.showMessageDialog(null, message);`

If Statements in a Nutshell

```
int letterCount = 0;
int upperCaseCount = 0;
String switchedCase = "";
for (int i = 0; i < message.length(); i++) {
    char nextChar = message.charAt(i);
    if (Character.isLetter(nextChar)) {
        letterCount++;
    }
    if (Character.isUpperCase(nextChar)) {
        upperCaseCount++;
        switchedCase += Character.toLowerCase(nextChar);
    } else if (Character.isLowerCase(nextChar)) {
        switchedCase += Character.toUpperCase(nextChar);
    } else {
        switchedCase += nextChar;
    }
}
```

Comparing Objects

- ▶ Exercise: **EmailValidator**
 - Use a **Scanner** object
 - Prompt for user's email address
 - Prompt for it again
 - Compare the two entries and report whether or not they match

 - ▶ Notice anything strange?
- 

Comparing Objects

- ▶ In Java:
 - `o1 == o2` compares *values*
 - `o1.equals(o2)` compares the internal state of *objects* (thus, their fields)

- ▶ Remember: variables of class type store **reference values**

- ▶ How should you compare the email addresses in the exercise?

Statement vs. Expressions

- ▶ Statements: used only for their *side effects*
 - Changes they make to stored values or control flow
- ▶ Expressions: calculate values
- ▶ Many statements contain expressions:
 - ```
if (amount <= balance) {
 balance -= amount;
} else {
 balance -= OVERDRAFT_FEE;
}
```

# Conditional Operator

- ▶ Let us choose between two possible values for an expression
- ▶ For example,
  - `balance -= (amount <= balance ? amount : OVERDRAFT_FEE);`
- ▶ is equivalent to:

```
if (amount <= balance) {
 balance -= amount;
} else {
 balance -= OVERDRAFT_FEE;
}
```
- ▶ Also called **ternary** or **selection** operator (Why?)

# Boolean Essentials—Like C

- ▶ Comparison operators: `<`, `<=`, `>`, `>=`, `!=`, `==`
- ▶ Comparing objects: `equals()`, `compareTo()`
- ▶ Boolean operators:
  - and: `&&`
  - or: `||`
  - not: `!`

# Predicate Methods

- ▶ A common pattern in Java:

```
public boolean isFoo() {
 ... // return true or false depending on
 // the Foo-ness of this object
}
```

# Test Coverage

- ▶ *Black box testing*: testing without regard to internal structure of program
  - For example, user testing
- ▶ *White box testing*: writing tests based on knowledge of how code is implemented
  - For example, unit testing
- ▶ *Test coverage*: the percentage of the source code executed by all the tests taken together
  - Want high test coverage
  - Low test coverage can happen when we miss branches of switch or if statements

# Switch and Enum

- »» The next five slides on switch and enumerations are optional. Do the Bid exercise if you're interested. See the book or the Google for more info. on switch and enum.

# Switch Statements: Choosing Between Several Alternatives

```
char grade = ...
int points;
switch (grade) {
case 'A':
 points = 95;
 break;
case 'B':
 points = 85;
 break;
...
default:
 points = 0;
}
```

Can switch on integer, character, or “enumerated constant”

Don't forget the breaks!

# Enumerated Constants

- ▶ Specify named sets:

```
public enum Suit {
 CLUBS, SPADES, DIAMONDS, HEARTS
}
```

- ▶ Store values from set:

```
Card c = new Card(2, CLUBS);
```

- ▶ Then switch on them:

```
switch (this.suit) {
 case CLUBS:
 case SPADES:
 return "black";
 default:
 return "red";
}
```

Why no break here?

Why no break here?

# Exercise: Bids for the Card Game “500”

```
switch (bidSuit) {
 case CLUBS:
 case SPADES:
 return “black”;
 default:
 return “red”;
}
```

- ▶ Implement a class Bid
  - Constructor should take a “trump” Suit and an integer representing a number of “tricks”
  - Test and implement a method, `getValue()`, that returns the point value of the bid, or 0 if the bid isn’t legal. See table for values of the legal bids.

|           | Spades | Clubs | Diamonds | Hearts | No Trump |
|-----------|--------|-------|----------|--------|----------|
| 6 tricks  | 40     | 60    | 80       | 100    | 120      |
| 7 tricks  | 140    | 160   | 180      | 200    | 220      |
| 8 tricks  | 240    | 260   | 280      | 300    | 320      |
| 9 tricks  | 340    | 360   | 380      | 400    | 420      |
| 10 tricks | 440    | 460   | 480      | 500    | 520      |

Suit enum is provided in the repository!

# Predicate Methods

- ▶ Live-coding:
  - Test and implement `isValid()` method for Bid
    - JUnit has test methods `assertTrue()` and `assertFalse()` that will be handy
  - Change `getValue()`: return 0 if `isValid()` is false

# Exercise

- ▶ Study your code for **Bid** and **BidTests**
- ▶ Do you have 100% test coverage of the methods?
  - **getValue()**
  - **isValid()**
- ▶ Add tests until you have 100% test coverage

# Work Time

- » Hand in quiz.
- Work on Homework 6: Grade and CubicPlot