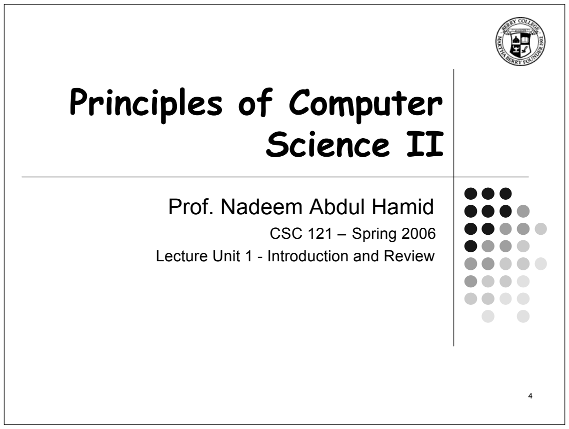
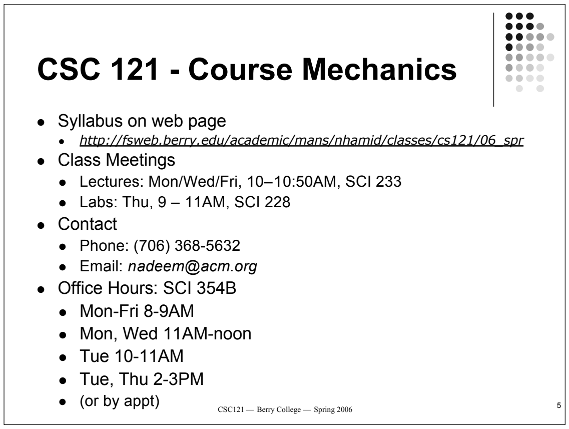


Principles of Computer Science II

Prof. Nadeem Abdul Hamid

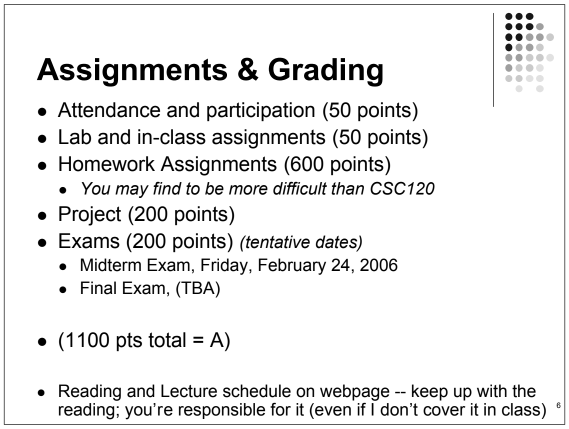
CSC 121 – Spring 2006

Lecture Unit 1 - Introduction and Review



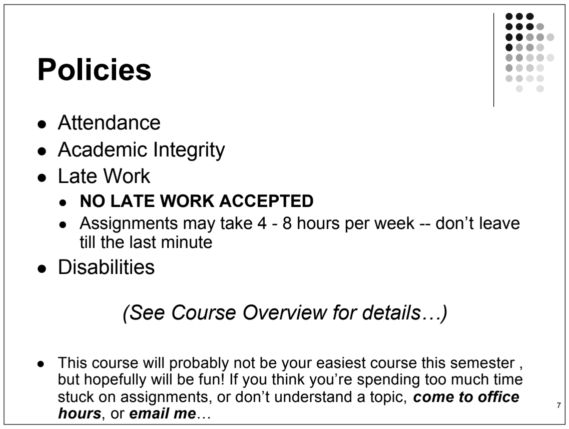
CSC 121 - Course Mechanics

- Syllabus on web page
  - *http://fsweb.berry.edu/academic/mans/nhamid/classes/cs121/06_spr*
- Class Meetings
  - Lectures: Mon/Wed/Fri, 10–10:50AM, SCI 233
  - Labs: Thu, 9 – 11AM, SCI 228
- Contact
  - Phone: (706) 368-5632
  - Email: *nadeem@acm.org*
- Office Hours: SCI 354B
  - Mon-Fri 8-9AM
  - Mon, Wed 11AM-noon
  - Tue 10-11AM
  - Tue, Thu 2-3PM
  - (or by appt)



Assignments & Grading

- Attendance and participation (50 points)
- Lab and in-class assignments (50 points)
- Homework Assignments (600 points)
  - *You may find to be more difficult than CSC120*
- Project (200 points)
- Exams (200 points) *(tentative dates)*
  - Midterm Exam, Friday, February 24, 2006
  - Final Exam, (TBA)
- (1100 pts total = A)
- Reading and Lecture schedule on webpage -- keep up with the reading; you're responsible for it (even if I don't cover it in class)

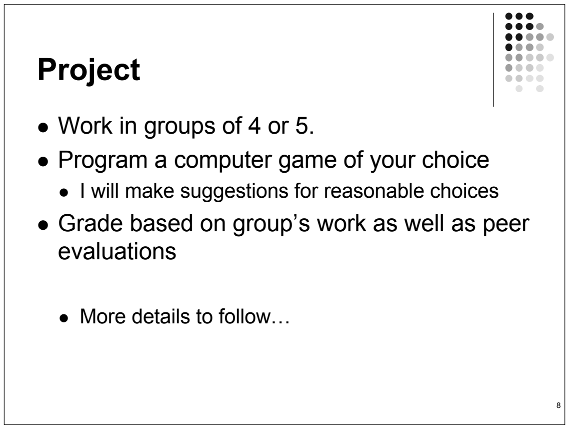


Policies

- Attendance
- Academic Integrity
- Late Work
  - **NO LATE WORK ACCEPTED**
  - Assignments may take 4 - 8 hours per week -- don't leave till the last minute
- Disabilities

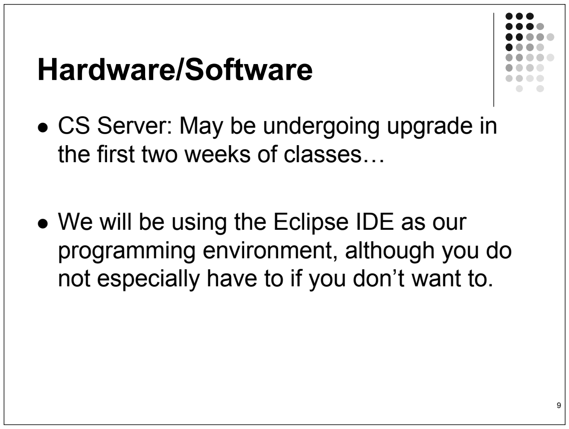
(See Course Overview for details…)

- This course will probably not be your easiest course this semester , but hopefully will be fun! If you think you're spending too much time stuck on assignments, or don't understand a topic, ***come to office hours***, or ***email me***…



Project

- Work in groups of 4 or 5.
- Program a computer game of your choice
  - I will make suggestions for reasonable choices
- Grade based on group's work as well as peer evaluations
  - More details to follow…



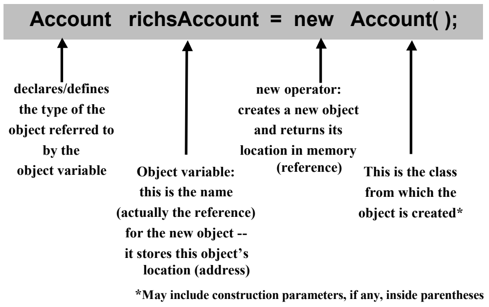
Hardware/Software

- CS Server: May be undergoing upgrade in the first two weeks of classes…
- We will be using the Eclipse IDE as our programming environment, although you do not especially have to if you don't want to.

Programming Review

- You should be familiar with…
  - Foundation of OO Programs: Classes, Objects, and Methods
  - Writing, Compiling, and Running Java Programs
  - Fundamental Data Types
  - Variables and Constants
  - Identifiers
  - Comments
  - Importing Packages and Classes
  - Basic Input/Output (I/O) Utilities
  - Explicit Type Conversion (“Typecasting” or “Casting”)
  - Selection
  - Loops

Instantiating a new object



Account richsAccount = new Account( );

declares/defines the type of the object referred to by the object variable

Object variable: this is the name (actually the reference) for the new object -- it stores this object’s location (address)

new operator: creates a new object and returns its location in memory (reference)

This is the class from which the object is created*

*May include construction parameters, if any, inside parentheses

Fundamental Data Types Summary

Data Type	Content	Memory Used	Range of Values
byte	[illegible]	[illegible]	[illegible]
short	[illegible]	[illegible]	[illegible]
int	[illegible]	[illegible]	[illegible]
long	[illegible]	[illegible]	[illegible]
float	[illegible]	[illegible]	[illegible]
double	[illegible]	[illegible]	[illegible]
char	[illegible]	[illegible]	[illegible]
boolean	[illegible]	[illegible]	[illegible]

Variable types in OO programs

- Local variables
- Parameters
- Instance fields/variables

- Primitive type variables
- Object references
- Constants (‘final’ variables)

Recommended access levels

- **Instance fields/variables → private**
  - This keeps the data “hidden” within the objects to support the OO concepts of ***encapsulation*** and ***information hiding***
- **Methods → public** (in some cases, *private*)
- **Constructors → public**
- **Classes → public** (in some cases, classes and their constructors may instead be *package*-level access)

- What if you forget to specify *public* or *private*?? The default is ***package*** access…

Explicit type conversion (casting)

- To perform an explicit type conversion (i.e., force a change in data from one type to another), use the **typecast** operator
- Example: to change an integer value, i, to float:
  (float) i
- Example: to change the calculated sum of two integers, *i* and *j*, to a float result:
  (float) (i + j)

- Avoids Java compilation errors due to narrowing conversions

Decisions/Selection

- A structured programming construct that enables the programmer to select between **two or more alternatives**, or “paths”, in a program
- The programmer must design the code so that decisions are reduced to “yes/no” style questions which can be evaluated as either:

  true or **false**
- For example: is *x* greater than *y*? Is *z* less than 2?
  - these are “yes/no” questions that evaluate to either true or false
- **boolean data types: true, false**

Relational operators

less than	**<**
less than or equal	**<=**
greater than	**>**
greater than or equal	**>=**
equal	**==**
not equal	**!=**

Logical operators

- Three logical operators are used to combine logical values for the purpose of making a complex decision:
  - ***not*** **!**
  - ***and*** **&&**
  - ***or*** **||**

Note: the order shown above also represents ***precedence for these logical operators***

Note: **!(x < y)** is logically the same as **x >= y**

Multiway selection

- Multiway selection chooses among several alternatives
- Two constructs:
  - ***switch*** statement (only used for evaluating selection conditions that are integer or character constants)
  - ***else if*** (alternate style to nested *if* statements – can evaluate selection conditions over a range of values)

switch **block example**

```
String input = JOptionPane.showInputDialog(“Enter selection (1 - 3): ”);
int someValue = Integer.parseInt(input);

switch (someValue)
{
   case 1: System.out.println(“This is case 1”);
           break;
   case 2: System.out.println(“This is case 2”);
           break;
   case 3: System.out.println(“This is case 3”);
           break;
   default: System.out.println(“This is default”);
            break;
}
```

Note: case labels may be integers or character constants (e.g., ‘A’)

else if source code example: what does this method do?

```
public void score2grade (double score)
{
    if (score >= 90.0)
      grade = 'A';
    else if (score >= 80.0)
      grade = 'B';
    else if (score >= 70.0)
      grade = 'C';
    else if (score >= 60.0)
      grade = 'D';
    else
      grade = 'F';
}
```

Comparing floating-point numbers

- Assume a very small number **ε** (Greek epsilon), typically declared as a miniscule value such as **10^{-14}**
- For practical purposes in a program's condition test, **consider two numbers equal if they are *close enough*** such that:

  $|x - y| <= ε$
- Or if dealing with really huge values, a better mathematical test for two values being close enough for equality is:

  $|x - y| / max(|x|, |y|) <= ε$
- Of course, if one of the values might be zero, don't divide by max(|*x* |, |*y*|), since that would make the denominator 0
- Based on your application, you might choose another **ε**

String comparison

- Do ***not*** use **==** to test if strings' contents are equal:
  if (input == "yes")   // WRONG!!!
- Instead, be sure to use the **equals( )** method:
  if (string1.equals(string2)) …
  if (input.equals("yes")) …
- Why? Because:
  - **==** tests if both items reference the ***same string object*** (tests if the ***reference***, or address location, is the ***same***)
  - **equals( )** tests if ***contents*** of both strings are the ***same***
- Useful case-insensitive test method ("Y" or "y")
  if (input.equalsIgnoreCase("Y"))

Lexicographic comparison of strings (continued)

- string1.compareTo(string2)
  - If it returns < 0: string1 alphabetizes before string2
  - If it returns > 0: string1 alphabetizes after string2
  - If it returns 0: string1 and string2 are equal
- Java's alphabetization rules:
  - "car" comes before "cargo"
  - "cargo" comes before "cathode"
  - Numbers come before letters (i.e., 8 comes before B)
  - Uppercase letters come before lowercase (i.e., "Hello" comes before "car")
  - Space character comes before all others

Object comparison: the same potential pitfall as strings

- Look at the following section of Java code

Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
Rectangle r = cerealBox;
Rectangle oatMealBox = new Rectangle(5, 10, 20, 30);

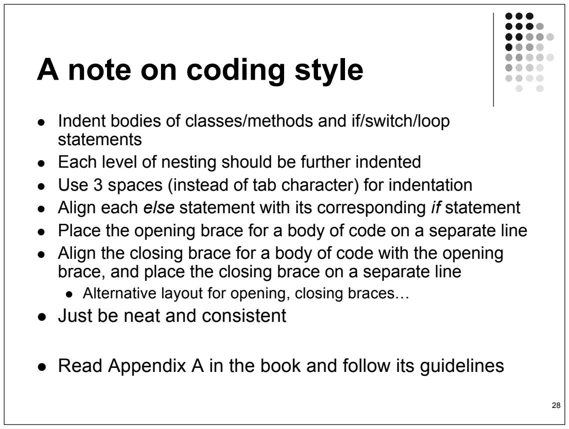
- This comparison will be ***true***:
  if (cerealBox == r) …
- But this comparison will be ***false***:
  if (cerealBox == oatMealBox) …
- What's going on here…

Object comparison: how to do it

- Do ***not*** use **==** to test if the ***contents*** of two separate rectangle objects are the same:
  if (cerealBox == oatmealBox)   // FALSE!!!
- Instead, use the **equals( )** method to do this:
  if (cerealBox.equals(oatMealBox))   //TRUE!!!
- Why? Because:
  - **==** tests if both ***references***, or object variables, ***refer to the same object*** (tests if the ***reference***, or address location, is the ***same*** for both → this tests for ***identity***)
  - **equals( )** tests if the ***contents*** of the rectangles are ***same***
- Later on, we'll learn that you must "***override***" the equals( ) method in a new class that you develop

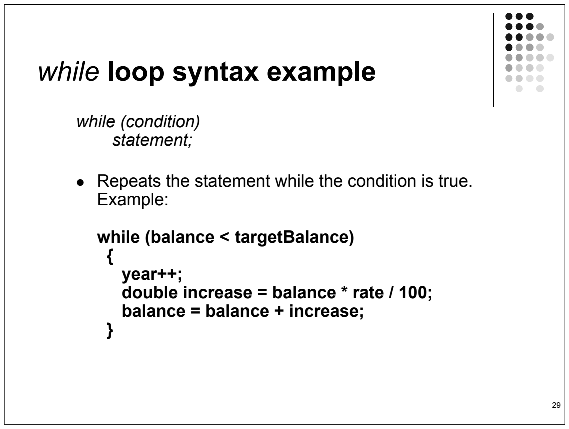
Testing for null reference

- An object variable (reference) can hold the value **null** -- it refers to no object (or no string) at all
- Use **==** in conditional tests to check for a null reference, for example:
  if (account == null)
- What good is this?? Here's an example:
  - **showInputDialog( )** returns **null** if the user hits the Cancel button of the input dialog window
    String input = JOptionPane.showInputDialog("...");
    if (input == null) { ... }   //user canceled dialog
- null is ***not*** the same as the empty string ""



A note on coding style

- Indent bodies of classes/methods and if/switch/loop statements
- Each level of nesting should be further indented
- Use 3 spaces (instead of tab character) for indentation
- Align each *else* statement with its corresponding *if* statement
- Place the opening brace for a body of code on a separate line
- Align the closing brace for a body of code with the opening brace, and place the closing brace on a separate line
  - Alternative layout for opening, closing braces…
- Just be neat and consistent
- Read Appendix A in the book and follow its guidelines

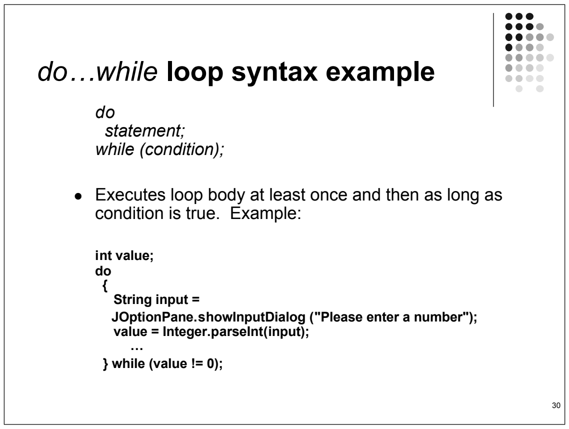


while loop syntax example

```
while (condition)
      statement;
```

- Repeats the statement while the condition is true. Example:

```
while (balance < targetBalance)
  {
     year++;
     double increase = balance * rate / 100;
     balance = balance + increase;
  }
```

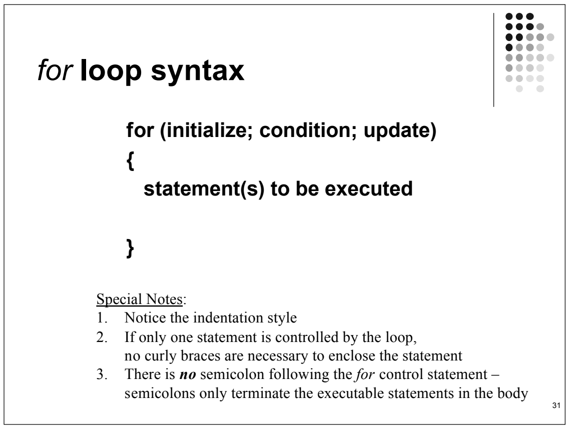


do…while loop syntax example

```
do
  statement;
while (condition);
```

- Executes loop body at least once and then as long as condition is true. Example:

```
int value;
do
 {
    String input =
    JOptionPane.showInputDialog ("Please enter a number");
    value = Integer.parseInt(input);
       …
 } while (value != 0);
```

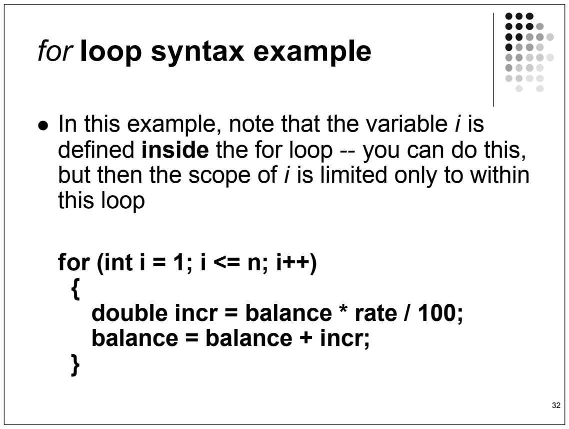


for loop syntax

```
for (initialize; condition; update)
{
   statement(s) to be executed


}
```

Special Notes:

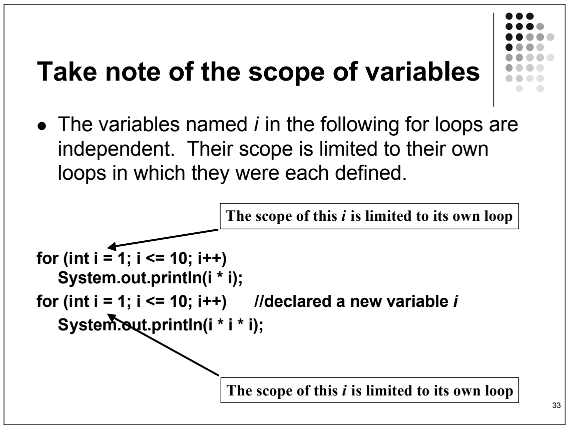
1. Notice the indentation style
2. If only one statement is controlled by the loop, no curly braces are necessary to enclose the statement
3. There is ***no*** semicolon following the *for* control statement – semicolons only terminate the executable statements in the body



for loop syntax example

- In this example, note that the variable *i* is defined **inside** the for loop -- you can do this, but then the scope of *i* is limited only to within this loop

```
for (int i = 1; i <= n; i++)
  {
     double incr = balance * rate / 100;
     balance = balance + incr;
  }
```



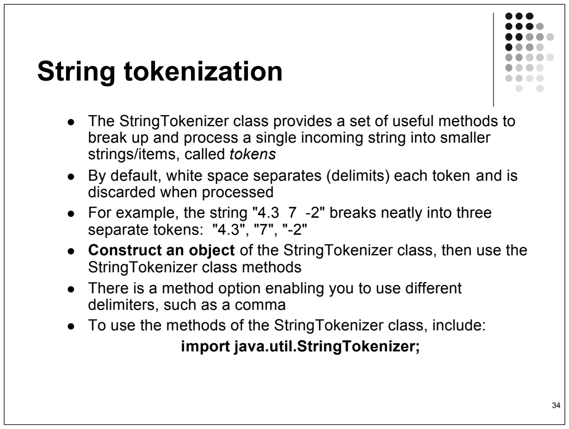
Take note of the scope of variables

- The variables named *i* in the following for loops are independent. Their scope is limited to their own loops in which they were each defined.

The scope of this *i* is limited to its own loop

```
for (int i = 1; i <= 10; i++)
   System.out.println(i * i);
for (int i = 1; i <= 10; i++)      //declared a new variable i
   System.out.println(i * i * i);
```

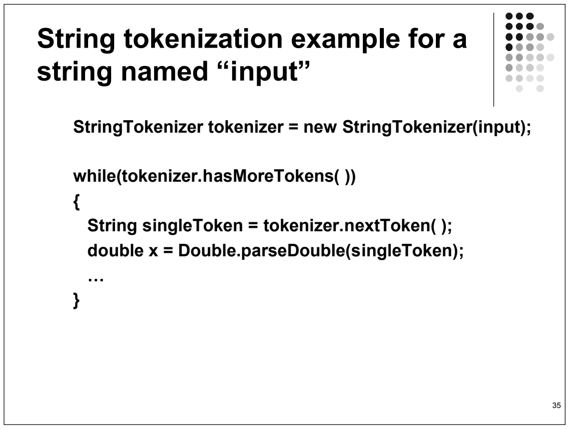
The scope of this *i* is limited to its own loop



String tokenization

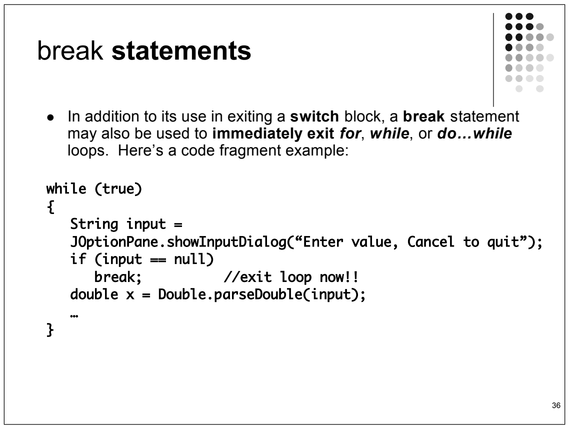
- The StringTokenizer class provides a set of useful methods to break up and process a single incoming string into smaller strings/items, called *tokens*
- By default, white space separates (delimits) each token and is discarded when processed
- For example, the string "4.3 7 -2" breaks neatly into three separate tokens: "4.3", "7", "-2"
- **Construct an object** of the StringTokenizer class, then use the StringTokenizer class methods
- There is a method option enabling you to use different delimiters, such as a comma
- To use the methods of the StringTokenizer class, include:

  import java.util.StringTokenizer;



String tokenization example for a string named “input”

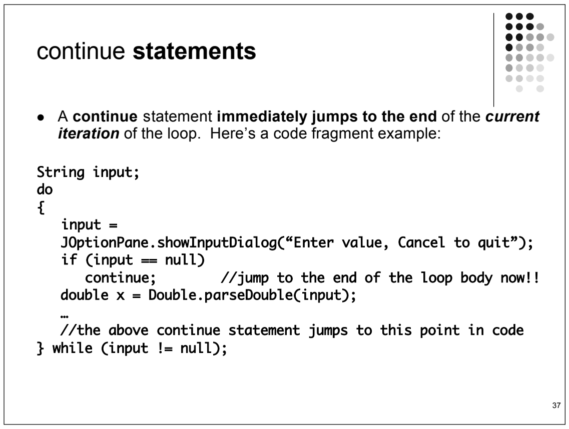
```
StringTokenizer tokenizer = new StringTokenizer(input);

while(tokenizer.hasMoreTokens( ))
{
  String singleToken = tokenizer.nextToken( );
  double x = Double.parseDouble(singleToken);
  …
}
```



break **statements**

- In addition to its use in exiting a **switch** block, a **break** statement may also be used to **immediately exit *for***, ***while***, or ***do…while*** loops. Here’s a code fragment example:

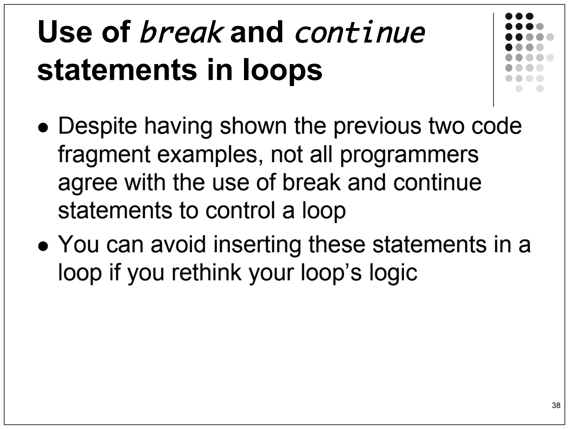
```
while (true)
{
   String input =
   JOptionPane.showInputDialog(“Enter value, Cancel to quit”);
   if (input == null)
      break;          //exit loop now!!
   double x = Double.parseDouble(input);
   …
}
```



continue **statements**

- A **continue** statement **immediately jumps to the end** of the ***current iteration*** of the loop. Here’s a code fragment example:

```
String input;
do
{
   input =
   JOptionPane.showInputDialog(“Enter value, Cancel to quit”);
   if (input == null)
      continue;        //jump to the end of the loop body now!!
   double x = Double.parseDouble(input);
   …
   //the above continue statement jumps to this point in code
} while (input != null);
```



Use of *break* and *continue* statements in loops

- Despite having shown the previous two code fragment examples, not all programmers agree with the use of break and continue statements to control a loop
- You can avoid inserting these statements in a loop if you rethink your loop’s logic