HST 952

Computing for Biomedical Scientists

Lecture 4

Outline

- Another look at Java built-in operators
- String and StringBuffer built-in java classes
- Classes, objects, and methods

Two Main Kinds of *Types* in Java

primitive data types

- the simplest types
- cannot decompose into other types
- have values only, no methods
- Examples:
  `int` - integer
  `double` - floating point
  `char` - character

class types

- more complex
- composed of other types (primitive or class types)
- have both data and methods
- Examples:
  `String`
  `StringBuffer`

Built-in Operators for primitive types

- Arithmetic (use with int, double, etc.):

  +, -, *, /, %

- Comparison (use with int, double, char, etc.):

  ==, !=, <, <=, >, >=

- Logical (use with boolean):

  &&, ||, !

Specialized Assignment Operators

- A shorthand notation for performing an operation on and assigning a new value to a variable
- General form: *`var <op>= expression;`*
  - equivalent to:

    `var = var <op> (expression);`
  - <op> is +, -, *, /, or %
- Examples:

```
amount += 25;
//amount = amount + 25;
```

Specialized Assignment Operators

```
amount *= 1 + interestRate;
 /*
 amount =
   amount * (1 + interestRate);
 */
```

- Note that the right side is treated as a unit (as though there are parentheses around the entire expression)

Increment and Decrement Operators

- Shorthand notation for common arithmetic operations on integer variables used for counting
- Some counters count up, some count down
- The counter can be incremented (or decremented) before or after using its current value

```
int count;
++count;  //preincrement count: count = count + 1 before using it
count++;  //postincrement count: count = count + 1 after using it
--count;  //predecrement count: count = count -1 before using it
count--;  //postdecrement count: count = count -1 after using it
```

Increment and Decrement Operators

Example:

```
        int x = 5;
        int y = 5;
        int result;
```

What will be the value of `result` after each of these executes? (assume each line is independent of the other)

```
        (a) result = x / ++y;
        (b) result = x / y++;
        (c) result = x + --y;
        (d) result = x + y--;
```

Returned Values

- Expressions *return* values: a number, character, etc. produced by an expression is “returned”, (it is the “return value.”)

```
int firstNumber, secondNumber,
productOfNumbers;
firstNumber = 5;
secondNumber = 9;
productOfNumbers = firstNumber *
secondNumber;
```

(in the last line, `firstNumber` returns the value 5 and `secondNumber` returns the value 9)

Returned Values

`firstNumber * secondNumber` is an expression that returns the integer value 45

- Similarly, methods return values

  `Integer.parseInt(str);` is a method of the Java built-in class Integer that returns the integer value of a string such as “12”, “67”, etc.

The String Class

- A string is a sequence of characters
- The String class is used to store strings
- The String class has methods to operate on strings
- String constant: one or more characters in *double* quotes
- Examples:

```
char charVariable = ‘a’
String stringVariable = "a";
String sentence = "Hello, world";
```

The String Class

- Individual characters in a variable of type String can be accessed *but not modified*
- To modify individual characters in a string, need to use a variable of type StringBuffer (more to come on class StringBuffer)
- A complete interface specification of Java's built-in classes and their methods (including that of the String class) is at:

  http://java.sun.com/j2se/1.3/docs/api/index.html

Indexing Characters within a String

- The index of a character within a string is an integer starting at 0 for the first character and gives the position of the character
- The `charAt`*`(Position)`* method returns the char at the specified position
- `substring`*`(Start, End)`* method returns the string from position *`Start`* to position *`End`*

Indexing Characters within a String

- Example:

  `String greeting = "Hi, there!";`

  `greeting.charAt(0)` returns `H`

  `greeting.charAt(2)` returns `,`

  `greeting.substring(4,6)` returns `the`

H	i	,		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8	9

The StringBuffer Class

- Implements a modifiable sequence of characters
  - the length and content of the sequence of characters can be modified using its methods
  - has many of the same methods as the String class and a few more (append, insert, replace)
- To create a new StringBuffer object that initially represents the string “rue” and assign it to a variable strBuffer, of type StringBuffer, write

```
StringBuffer strBuffer = new StringBuffer(“rue”);
// illegal to write StringBuffer strBuffer = “rue”
```

or write

```
String str = “rue”;
StringBuffer strBuffer = new StringBuffer(str);
```

The StringBuffer Class

- Modify the sequence:

```
strBuffer.append(‘s’);
System.out.println(strBuffer);  // prints out rues
System.out.println(strBuffer.length());  // prints out 4
strBuffer.insert(2, ‘s’);
System.out.println(strBuffer); // prints out ruses
strBuffer.insert(1, “ef”);
System.out.println(strBuffer); // prints out refuses
System.out.println(strBuffer.length());  // prints out 7
strBuffer.replace(2, 3, “-”);
System.out.println(strBuffer); // prints out re-uses
```

Classes, Objects, and Methods

- Instance variables
- Instantiating (creating) objects
- A look at methods
- Parameter passing (pass-by-value and pass-by-reference)
- Static methods and static variables

Instance Variables (Data Items)

- Person class has the following instance variables/data items: firstName, lastName, and age:

```
public String firstName;

public String lastName;

public double age;
```

- `public` means that there are no restrictions on how an instance variable is used
- `private` means that the instance variable cannot be accessed directly outside the class
- In general, instance variables should be declared `private` instead of `public`

Instance Variables (Data Items)

```
public class Person
{
    private String firstName;
    private String lastName;
    public double age;
    public String getFirstName()
    {
        return(firstName);
    }
// other method definitions ...
}
```

Instantiating (Creating) Objects

- Syntax:

```
ClassName instanceName =
                    new ClassName();
```

- Note the keyword *new*
- Example: instantiate an object of class Person within the definition of another class

```
Person newPerson = new Person();
```

- Public instance variables can be accessed and modified using the dot operator:

```
newPerson.age = 35.5;
```

Instantiating (Creating) Objects

- Private instance variables cannot be modified/accessed in this way:

```
newPerson.firstName = "B'Elanna"; //illegal
```

- Define public get and set methods in class Person to retrieve and modify values of private instance variables:
  - `public String getFirstName()`
  - `public void setFirstName(String fName)`
  - `public String getLastName()`
  - `public void setLastName(String lName)`
- To set first and last name instance variables:
  - `newPerson.setFirstName("B'Elanna");`
  - `newPerson.setLastName("Torres");`

Instantiating (Creating) Objects

- To retrieve values of first and last name instance variables:
  - `newPerson.getFirstName();`
    `//returns “B’Elanna”`
  - `newPerson.getLastName();`
    `//returns “Torres”`
- Instance variable age should also be private:
  - `private double age;`
  - `public double getAge()`
  - `public void setAge(double ageValue)`

Return Type of Methods

- As seen in previous slides, some methods perform an action *and return a single value*
- Some methods just perform an action (e.g. print a message) and do not return a value
- All methods require that the return type be specified
- Return types may be:
  - a primitive data type, such as `char`, `int`, `double`
  - a class, such as `String`, `Person`, etc.
  - `void` if no value is returned

Return Type of Methods

- You can use a method wherever it is legal to use its return type, for example the `getFirstName()` method of `Person` returns a String, so this is legal:

```
Person anotherPerson =
   new Person();
String name =
   anotherPerson.getFirstName();
```

- Also legal:

```
double age =
   anotherPerson.getAge();
```

Return Statement

- Methods that return a value must execute a `return` statement that includes the value to return
- For example:

```
public double getAge()
{
 return age;
  //return(age); could be used instead
}
private double age = 79.6;
```

- A return statement is not required in a method that does not return a value (has a `void` return type)

Good Programming Practice

- Start class names with a capital letter
- Start method names with a lower case letter
- Include comments in your code that describe
  - what each class does
  - what each method does
  - any unusual/non-intuitive steps taken in solving a problem

The main Method

- A program written to solve a problem (rather than define an object) is written as a class with one method, `main`
- Invoking the class name invokes the `main` method
- Example: `HelloWorld Class`
- Note the basic structure:

```
public class HelloWorld
{
  public static void main(String[] args)
   {
    <statements that define the main method>
   }
}
```

The "this." Operator

- `this.` refers to the object that contains the reference (an object's way of referring to itself)
- Methods called in a .java file that gives an object's definition do not need to reference the object
- In such files, you may omit the use of "`this.`" in referring to a method, since it is presumed
- For example, if `answerOne()` is a method defined in the class `Oracle`:

The "this." Operator

```
public class Oracle
{
  private int firstNum = 5;
  private int secondNum = 10;

  public int answerOne()
  {
    return(firstNum + secondNum);
  }

  // code stored in file Oracle.java
```

The "this." Operator

```
  public int getAnswer()
  {
   /* One way to invoke the answerOne
      method defined in this file
      (Oracle.java)is: answerOne();
   */
   //Another way is to use "this."
   int num = this.answerOne();
   return(num);
  }

} // end class Oracle
```

Calling an Object’s Methods

- To call a method *outside* its object definition file, in general, a valid object name should precede the method name
- For example (in a file other than Oracle.java):

```
Oracle myOracle = new Oracle();
//myOracle is not part of the definition
//code for Oracle
...
//dialog is a method defined in Oracle class
myOracle.dialog();
```

Local Variables and *Blocks*

- A *block* (also called a *compound statement)* is the set of statements between a pair of matching braces (curly brackets)
- A variable declared inside a block is known only inside that block
  - it is *local* to the block, therefore it is called a *local variable*
  - when the block finishes executing, local variables disappear
  - references to it outside the block cause a compile error

Local Variables and *Blocks*

- Some programming languages (e.g. C and C++) allow a variable’s name to be reused outside the local block
  - – this is confusing and not recommended
- In Java, a variable name can be declared only *once for a method*
  - – although the variable does not exist outside the local block, other blocks in the same method cannot reuse the variable's name

Variable Declaration

- Declaring variables outside all blocks but within a method definition makes them available within all the blocks in that method:

```
public void printSomeValue(int n)
{
  int i=0; // i  is available in all blocks (including if and while)
  if (i < n) {
        int j = (i + n) * 50; // j is available only in the if block;
  }
  while (j < 50) {  // illegal, j is not available outside if block
     System.out.println(“j is ” + j);
     j++;
  }
}
```

Variable Declaration

Good Programming Practice:

- declare variables just before you use them
- initialize variables when you declare them
- do not declare variables inside loops
  - it takes time during execution to create and destroy variables, so it is better to do it just once for loops
- it is okay to declare loop counters in the *`Initialization`* field of `for` loops, e.g. `for(`**`int i=0;`**` i <10; i++)`...
  - the *`Initialization`* field executes only once, when the `for` loop is first entered

Passing Values to a Method: Parameters

- Some methods can be more flexible (and useful) if we pass them input values
- Input values for methods are called *passed* values or *parameters*
- Parameters and their data types must be specified inside the parentheses of the heading in the method definition
  - these are called *formal* parameters
- The calling object must put values of the same data type, in the same order, inside the parentheses of the method invocation
  - these are called *arguments*, or *actual* parameters

Parameter Passing Example

```
//Definition of method to double an integer
public int doubleValue(int numberIn)
{
   return 2 * numberIn;
}
//Invocation of the method... somewhere in main...
int next = 55;
System.out.println("Twice next = " + doubleValue(next));
```

- Formal parameter in the method definition:
  - `numberIn`
- Argument in the method invocation:
  - `next`

Pass-By-Value: *Primitive* Data Type Arguments

- When the method is called, the *value* of each argument is *copied* (assigned) to its corresponding formal parameter
- The number of arguments must be the same as the number of formal parameters
- The data types of the arguments must be the same as the formal parameters and in the same order

Pass-By-Value: *Primitive* Data Type Arguments

- Formal parameters are initialized to the values passed
- Formal parameters are local to the method for which they are defined
- Variables used as arguments cannot be changed by the method
  - the method only gets a copy of the variable's value

Variables: Class Type vs. Primitive Type

What does a variable hold?

- It depends on whether its type is a *primitive* type or *class* type

- A primitive type variable holds the value of the variable

- Class types are more complicated
  - classes have methods and instance variables

Variables: Class Type vs. Primitive Type

- A class type variable holds the *memory address* of the object
  - the variable does not actually hold the value of the object
  - in fact, as stated above, objects generally do not have a single value and they also have methods, so it does not make sense to talk about an object’s "value"

Variables: Class Type vs. Primitive Type

- See handout

Assignment with Variables of a Class Type

```
klingon.set(“Klingon ox”, 10, 15);
earth.set(“Black rhino”, 11, 2);
earth = klingon;
earth.set(“Elephant”, 100, 12);
System.out.println(“earth:”);
earth.writeOutput();
System.out.println(“klingon:”);
klingon.writeOutput();
```

What will the output be?

(see the next slide)

Assignment with Variables of a Class Type

```
klingon.set(“Klingon ox”, 10, 15);
earth.set(“Black rhino”, 11, 2);
earth = klingon;
earth.set(“Elephant”, 100, 12);
System.out.println(“earth:”);
earth.writeOutput();
System.out.println(“klingon:”);
klingon.writeOutput();
```

What will the output be?

`klingon` and `earth` both print elephant.

Why do they print the same thing?

(see the next slide)

Output:

```
earth:
Name = Elephant
Population = 100
Growth Rate = 12%
klingon:
Name = Elephant
Population = 100
Growth Rate = 12%
```

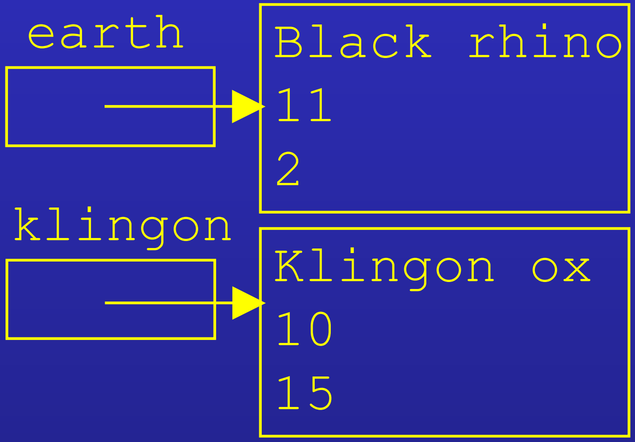
```
klingon.set("Klingon ox", 10, 15);
earth.set("Black rhino", 11, 2);
earth = klingon;
earth.set("Elephant", 100, 12);
System.out.println("earth:");
earth.writeOutput();
System.out.println("klingon:");
klingon.writeOutput();
```

Why do they print the same thing?

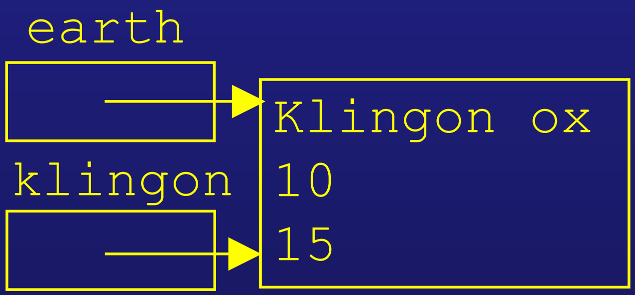
The assignment statement makes earth and klingon refer to the same object.

When earth is changed to "Elephant", klingon is changed also.

Before the assignment statement, earth and klingon refer to two different objects.



After the assignment statement, earth and klingon refer to the same object.



Assignment with Variables of a Class Type

- A class variable returns a number corresponding to the *memory address* where the object with that variable name is stored
- If two class variables are compared using ==, it is their addresses, not their values that are compared!
- This is rarely what you want to do!
- Use the class's `.equals()` method to compare the *values* of class variables

Comparing Class Variables

```
Person firstPerson = new Person();
firstPerson.setFirstName(“Lisa”);
Person secondPerson = new Person();
secondPerson.setFirstName(“Barry”);

if(firstPerson == secondPerson)
//this compares their addresses□
{
   <body of if statement>
}

if(firstPerson.equals(secondPerson)
//this compares their variable values□
{
   <body of if statement>
}
```

Pass-by-Reference:
Class Types as Arguments

- Class variable names used as parameters in a method call copy the argument's ***address***□ (not the value) to the formal parameter
- So the formal parameter name also contains the address of the argument
- It is as if the formal parameter name is an alias for the argument name

Pass-by-Reference: *Class* Types as Arguments

- Any action taken on the formal parameter is actually taken on the original argument

- Unlike the situation with primitive types, the original argument is *not* protected for class types

Class Type as a Method Argument

```
//Method definition with a DemoSpecies class
//parameter
public void makeEqual(DemoSpecies otherObject)
{
   otherObject.name = this.name;
   otherObject.population =
     this.population;
   otherObject.growthRate =
     this.growthRate;
}

//Method invocation
DemoSpecies s1 = new
     DemoSpecies("Crepek", 10, 20);
DemoSpecies s2 = new DemoSpecies();
s1.makeEqual(s2);
```

Class Type as a Method Argument

```
//Method definition with a DemoSpecies class parameter
public void makeEqual(DemoSpecies otherObject)
{
   otherObject.name = this.name;
   otherObject.population = this.population;
   otherObject.growthRate = this.growthRate;
}

//Method invocation
DemoSpecies s1 = new DemoSpecies("Crepek", 10, 20);
DemoSpecies s2 = new DemoSpecies();
s1.makeEqual(s2);
```

- The method call makes `otherObject` an alias for `s2`, therefore *the method acts on* `s2`*, the* `DemoSpecies` *object passed to the method!*

- This is *unlike* primitive types, where the passed variable cannot be changed.

Static Methods

- Sometimes there is no obvious object to which a method should belong (e.g., a method to compute the square root of a number)
- Use the static keyword in defining such methods
- Static methods can be called without first creating an object
- Use the class name instead of an object name to invoke them
- Static methods are also called *class methods*

Static Methods

- Declare static methods with the `static` modifier, for example:

  public static double circleArea(double radius) ...

- Since a static method doesn't need a calling object, it cannot refer to a (nonstatic) instance variable of its class.
- Likewise, a static method cannot call a nonstatic method of its class (unless it creates an object of the class to use as a calling object).

Uses for Static Methods

- Static methods are commonly used to provide libraries of useful and related functions
- Examples:
  - The different read methods in the SavitchIn class`
  - the Math class
    - automatically provided with Java
    - functions include pow, sqrt, max, min, etc.
    - more details to come

The Math Class

- Includes constants `Math.PI` (approximately 3.14159) and `Math.E` (base of natural logarithms which is approximately 2.72)
- Includes three similar static methods: `round`, `floor`, and `ceil`
  - All three return whole numbers (although they are of type `double`)
  - **`Math.round`** returns the whole number nearest its argument

The Math Class

`Math.round(3.3)` returns `3.0` and
`Math.round(3.7)` returns `4.0`

- **`Math.floor`** returns the nearest whole number that is equal to or less than its argument

`Math.floor(3.3)` returns `3.0` and
`Math.floor(3.7)` returns `3.0`

- **`Math.ceil`** (short for ceiling) returns the nearest whole number that is equal to or greater than its argument

`Math.ceil(3.3)` returns `4.0` and
`Math.ceil(3.7)` returns `4.0`

Static Variables

- Example of a static variable definition:

```
private static int numTries = 0;
```

- Similar to definition of a named constant, which is a special case of static variables.
- Static variables may be public or private but are usually private for the same reasons instance variables are.

Static Variables

- Only one copy of a static variable exists for a class and it can be accessed by any object of the class.
- May be initialized (as in example above) or not.
- Can be used to let objects of the same class coordinate (see 2nd handout).

Read

- Chapter 4