**Oriette Ramos**

**Lesson Plan 34**

**ISTA-220**

1. Explain the difference between the concepts of associativity and precedence.  
   **Each operator has a precedence. For example, the \* operator has a higher precedence than the + operator. This means that the expression a + b \* c is the same as a + (b \* c). Each operator also has an associativity that defines whether the operator evaluates from left to right or from right to left. For example, the = operator is right-associative (it evaluates from right to left), so a = b = c is the same as a = (b = c).**
2. Explain the difference between the concepts of binary and unary operators.  
   **A unary operator is an operator that has just one operand. For example, the increment operator (++) is a unary operator. A binary operator is an operator that has two operands. For example, the multiplication operator (\*) is a binary operator.**
3. List four constraints imposed by C# with respect to operator overloading.
   * **You cannot change the precedence and associativity of an operator. Precedence and associativity are based on the operator symbol (for example, +) and not on the type (for example, int) on which the operator symbol is being used. Hence, the expression a + b \* c is always the same as a + (b \* c) regardless of the types of a, b, and c.**
   * **You cannot change the multiplicity (the number of operands) of an operator. For example, \* (the symbol for multiplication) is a binary operator. If you declare a \* operator for your own type, it must be a binary operator.**
   * **You cannot invent new operator symbols. For example, you can’t create an operator symbol such as \*\* for raising one number to the power of another number. You’d have to define a method to do that.**
   * **You can’t change the meaning of operators when they are applied to built-in types. For example, the expression 1 + 2 has a predefined meaning, and you’re not allowed to override this meaning. If you could do this, things would be too complicated.**
   * **There are some operator symbols that you can’t overload. For example, you can’t overload the dot (.) operator, which indicates access to a class member. Again, if you could do this, it would lead to unnecessary complexity.**
4. What is the syntax for overloading operators? Discuss access, scope, return value types, and parameter types and multiplicity.  
   **You use method-like syntax with a return type and parameters, but the name of the method is the keyword operator together with the operator symbol you are declaring. All operators must be public. All operators must be static. Operators are never polymorphic and cannot use the virtual, abstract, override, or sealed modifiers. A binary operator has two explicit arguments, and a unary operator has one explicit argument. At least one of the parameters must always be of the containing type.**
5. What are symmetric overloaded binary operators and how do they differ from non-symmetric overloaded binary operators?  
   **With symmetric overloaded binary operators, it does not matter what the order of the operands; a + b == b + a. With non-symmetric overloaded binary operators, you must ensure the the operands are the correct type based on position.**
6. Can you overload compound assignment operators? If so, please state how you do so. If not, explain why not.  
   **You cannot specifically overload a compound assignment operator. A compound assignment operator (such as +=) is always evaluated in terms of its associated simple operator (such as +). In other words, the statement a += b; is automatically evaluated like this: a = a + b;. If you have overloaded the appropriate simple operator, the overloaded version is automatically called when you use its associated compound assignment operator.**
7. What is the difference between overloading increment and decrement operators for value types and reference types?  
   **Value types can easily be written to increment the value, but care must be taken with reference types to avoid copying the reference. Instead, a new object must be created based on the data in the original. This can become costly in terms of memory and garbage-collection overhead.**
8. Why do we overload some operators in pairs?  
   **Some operators naturally come in pairs. For example, if you can compare two values by using the != operator, you would expect to be able to also compare the two values by using the == operator. The C# compiler enforces this very reasonable expectation by insisting that if you define either operator == or operator !=, you must define them both. This neither-or-both rule also applies to the < and > operators and the <= and >= operators. The C# compiler does not write any of these operator partners for you. You must write them all explicitly yourself, regardless of how obvious they might seem..**
9. What is the difference between widening conversion and narrowing conversion?  
   **An implicit conversion is sometimes called a widening conversion because the result is wider than the original value—it contains at least as much information as the original value, and nothing is lost. In the case of int and double, the range of double is greater than that of int, and all int values have an equivalent double value. However, the converse is not true, and a double value cannot be implicitly converted to an int. A double can be converted to an int, but the conversion requires an explicit notation (a cast). An explicit conversion is sometimes called a narrowing conversion because the result is narrower than the original value (that is, it can contain less information) and can throw an OverflowException exception if the resulting value is out of the range of the target type.**
10. What is the difference between explicit conversion and implicit conversion?  
    **An explicit conversion requires explicit notation (a cast) and can through an verflowException exception if the resulting value is out of the range of the target type. An implicit conversion requires no special syntax and never throws an exception.**