**Java Class 10**

**Writing Classes and Using Objects**

True **object-oriented programming** is based on defining classes that represent objects with well-defined characteristics and functionality

The programs we’ve written in previous examples have used classes defined in the Java Standard Class Library

Now we will begin to design programs that rely on classes that we will write ourselves

The class that contains the main method is just the starting point of a program and is sometimes referred to as the **driver**

The reserved word class means “I’m about to tell you what a new type of object looks like”

An object that we create from a class has attributes/state (data) and operations/behavior (methods)

**Classes and Objects**

Consider a six-sided die (singular of dice)

◦ We represent a die by designing a class called Die that models this state and behavior

◦ It’s state can be defined as which face is showing

◦ It’s primary behavior is that it can be rolled

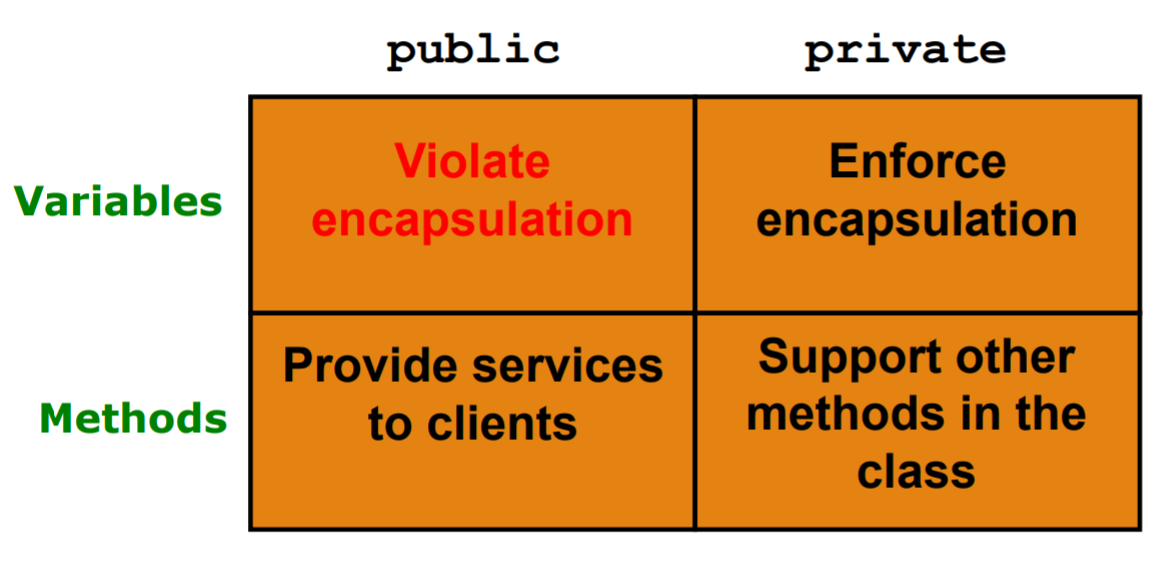
Once we have created a Die class, we can instantiate as many die objects as we need for any particular program

**Classes**

A class can contain Instance data declarations and method declarations

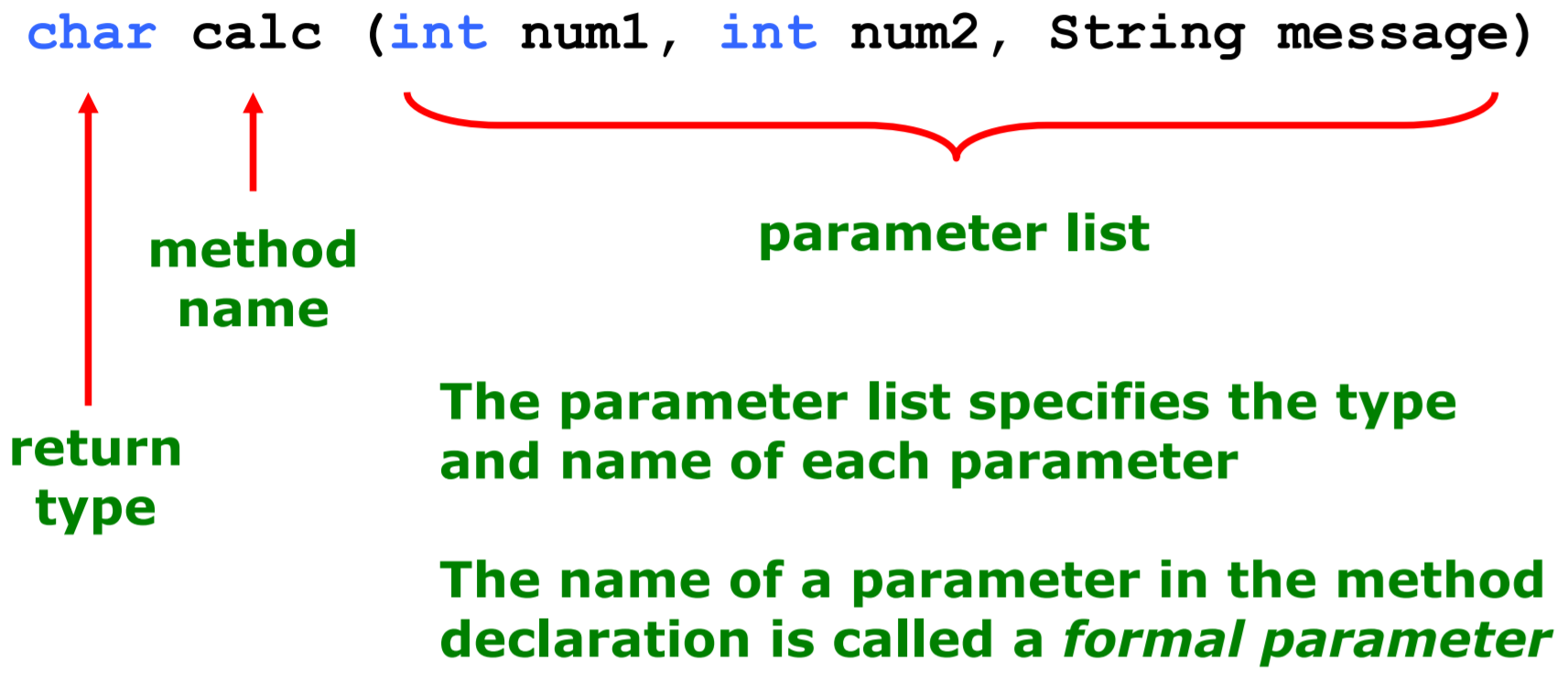
* Instance Data
  + A variable declared at the class level (such as faceValue) is called **instance data**
  + Each instance of an object has its own set of instance variables
  + That's the only way two objects can have different states
  + Each time a Die object is created, a new faceValue variable is created for this particular object, and it is given an initial value by a special method called the **constructor**
    - The **constructor** has the same name as the class and creates the initial state of an object
  + Even though each object of a class has its own data space, the objects share the method definitions
  + We can depict the two Die objects from the RollingDice program as follows:
    - Each object maintains its own faceValue variable, and thus its own state
* Dog Class
  + Each class declares:
    - Instance variables: descriptive characteristics or states of the object (size and name of a dog)
      * A variable declared at the class level (such as size) is called an instance variable
      * Each time a Dog object is created, a new memory space is reserved for variable size, name and breed in this particular object
      * The objects of a class each have their own set of (different) values for their instance variables, and thus each has its own state
      * At the same time the objects of a class share the same method definitions
      * The **scope** of a variable is the area in a program in which that variable can be referenced (used)
      * Variables declared at the class level can be referenced by all methods in that class
      * **Instance vs. Local Variables**
        + Instance variables are declared inside a class but not within a method. Lifetime: instance variables exist as long as the object exists. Scope: instance variables can be referenced by all methods in that class
        + Local variables are declared with a method. Lifetime: when the method finishes, all local variables are destroyed (including the parameters of the method). Scope: local variables can only be used in that method
        + Instance variables always get a default value (but don’t rely on it). If you don’t explicitly assign a value to an instance variable, the instance variable still has a value. Integers: 0 floating points: 0.0 booleans: false references: null
        + Local variables must be initialized before use
    - Methods: behaviors of the object or what the object can do (bark)

**Encapsulation**

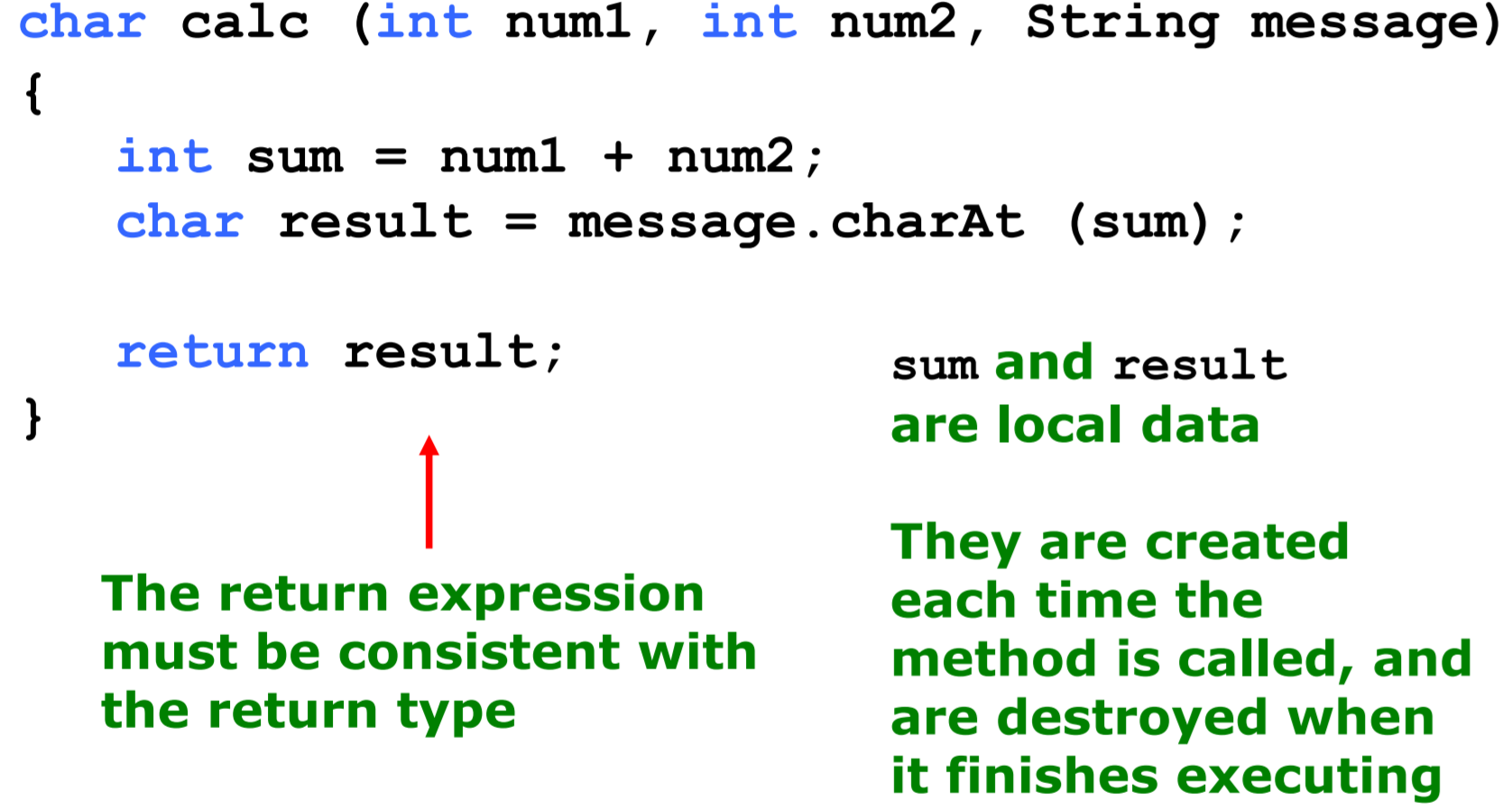
* An encapsulated object can be thought of as a black box -- its inner workings are hidden from the client (class that uses the object)
* The client object invokes the interface methods and they manage the instance data
* **Visibility Modifiers**
  + In Java, we accomplish encapsulation through the appropriate use of **visibility modifiers**
  + A *modifier* is a Java reserved word that specifies particular characteristics of a method or data
  + We've used the final modifier to define constants
  + Java has three visibility modifiers: public, protected, and private
  + Members of a class that are declared with **public visibility** can be referenced anywhere and anyone can use them
  + Members of a class that are declared with **private visibility** can be referenced only within that class and no one else can use them
  + Members declared without a visibility modifier have **default visibility** and can be referenced by any class in the same package but outside the package those same members appear as private
  + Public variables violate encapsulation because they allow the client to modify the values directly
  + Therefore instance variables should be declared with private visibility
  + It is acceptable to give a constant public visibility, which allows it to be used outside of the class
  + Public constants do not violate encapsulation because, although the client can access them, their values cannot be changed
  + 
* **Accessors and Mutators**
  + Because instance data are private, a class usually provides services to access and modify data values
  + An **accessor** method returns the current value of a variable
  + A **mutator** method changes the value of a variable
  + The names of accessor and mutator methods take the form getX and setX, respectively, where X is the name of the value
  + They are sometimes called “getters” and “setters”
  + **Mutator Restrictions**
    - The use of mutators gives the class designer the ability to restrict a client’s options to modify an object’s state
    - A mutator is often designed so that the values of variables can be set only within particular limits
    - For example, the setFaceValue mutator of the Die class should restrict the value to the valid range (1 to MAX)

**Java Class 11**

**How To Define A Method**

* A method declaration (definition) begins with a **method header**
* 

**Method Body**

* The method header is followed by the method body
* 
* **The return Statement**
  + The **return type** of a method indicates the type of value that the method sends back to the calling location
  + A method that does not return a value has a void return type
  + A **return statement** specifies the value that will be returned (return expression;)
  + Its expression must conform to the return type
* **Parameters**
  + When a method is called, the **actual parameters** in the invocation are copied into the **formal parameters** in the method header
* **Scope and Local Data**
  + As we’ve seen, local variables can be declared inside a method
  + Keep in mind that instance variables, declared at the class level, exist as long as the object exists
  + The formal parameters of a method create automatic local variables when the method is invoked
  + When the method finishes, all local variables are destroyed (including the formal parameters)
* **Method Control Flow**
  + If the called method is in the same class, only the method name is needed
  + The called method is often part of another class or object. When that happens, the method is called using the class or object name, dot operator, and the method name

**Constructors Revisited**

* Are we calling a method named Duck? Not really, we are calling the Duck constructor
* The constructor runs when you instantiate an object when using the keyword new followed by the class name
* If a class has no constructor (such as the Dog class), the **compiler** automatically creates a default constructor (public Dog() { }) 🡨 default constructor
* How is a constructor different from a regular method?
  + The name of a constructor is the **same** as the class name
  + A constructor cannot return a value and does not have a return type (it is even not a void return type)
* A common usage of a constructor: to initialize the state (instance variables) of an object with parameters (arguments)

**Driver Programs**

* A **driver program** drives the use of other, more interesting parts of a program
* Driver programs are often used to test other parts of the software
* The Transactions class contains a main method that drives the use of the Account class, exercising its services

**Quick Check**

* How do we express which Account object's balance is updated when a deposit is made?
* Each account is referenced by an object reference variable:
* Account myAcct = new Account(…);
* When a method is called, you call it through a particular object:
* myAcct.deposit(50);

**Java Class 12**

**Arcs**

* In JavaFX, an arc is defined as a portion of an ellipse
* Like an ellipse, the first four parameters to the Arc constructor specify the center point (x and y) as well as the radii along the horizontal and vertical
* Two additional parameters specify the portion of the ellipse that define the arc
* The Arc constructor:
* Arc(centerX, centerY, radiusX, radiusY, startAngle, arcLength)
* The start angle is where the arc begins relative to the horizontal
* The arc length is the angle that defines how big the arc is
* Both angles are specified in degrees
* An arc also has an arc type:
  + ArcType.OPEN The curve along the ellipse edge
  + ArcType.CHORD End points are connected by a straight line
  + ArcType.ROUND End points are connected to the center point of the ellipse, forming a rounded “pie” piece
* The start angle or the arc length could have been specified using negative values. If negative, the angle is measured clockwise instead of counterclockwise.

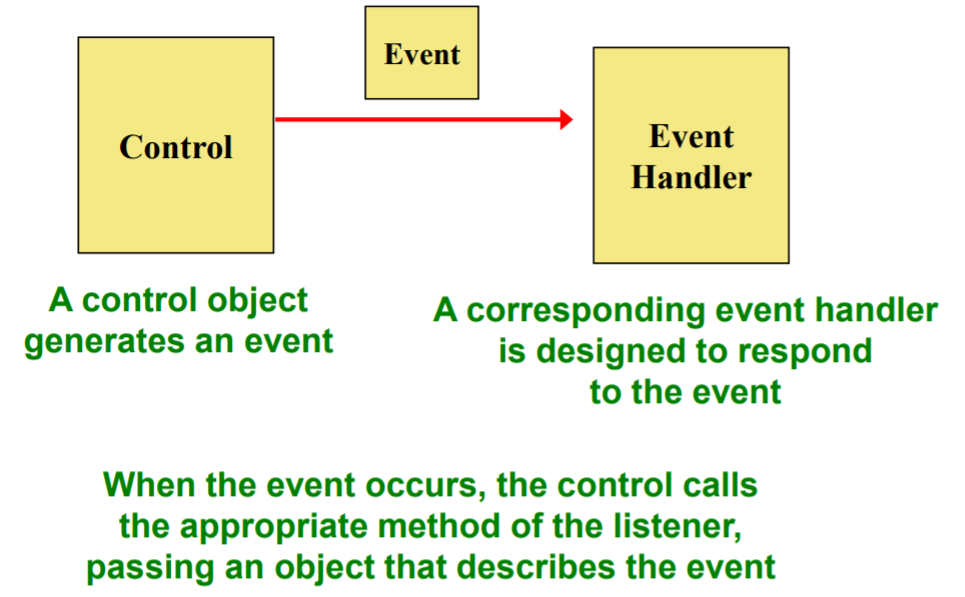
**Images**

* The JavaFX Image class is used to load an image from a file or URL
* Supported formats: jpeg, gif, and png
* To display an image, use an ImageView object
* An Image object cannot be added to a container directly
* The parameter to the Image constructor can include a pathname:
* Image logo = new Image("myPix/smallLogo.png");
* It can also be a URL: Image logo = new("http://example.com/images/bio.jpg");
* **Viewports**
  + A viewport is a rectangular area that restricts the pixels displayed in an ImageView
  + It is defined by a Rectangle2D object:
  + imgView.setViewport(new Rectangle2D(200, 80, 70, 60));

**Graphical User Interfaces**

* A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
  + controls, events, and event handlers
* A ***control*** is a screen element that displays information or allows the user to interact with the program: ◦ labels, buttons, text fields, sliders, etc.
* An ***event*** is an object that represents some activity to which we may want to respond

For example, we may want our program to perform some action when the following occurs:

* + a graphical button is pressed
  + a slider is dragged
  + the mouse is moved
  + the mouse is dragged
  + the mouse button is clicked
  + a keyboard key is pressed
* The Java API contains several classes that represent typical events
* Controls, such as a button, generate (or fire) an event when it occurs
* We set up an ***event handler*** object to respond to an event when it occurs
* We design event handlers to take whatever actions are appropriate when an event occurs
* 
* A JavaFX button is defined by the Button class
* It generates an action event
* The PushCounter example displays a button that increments a counter each time it is pushed
* A call to the setOnAction method sets up the relationship between the button that generates the event and the event handler that responds to it
* This example uses a *method reference* (using the :: operator) to specify the event handler method
* The *this* reference indicates that the event handler method is in the same class
* So the PushCounter class also represents the event handler for this program
* The event handler method can be called whatever you want, but must accept an ActionEvent object as a parameter
* In this example, the event handler method increments the counter and updates the text object
* The counter and Text object are declared at the class level so that both methods can use them
* In this example, a FlowPane is used as the root node of the scene
* A flow pane is another layout pane, which displays its contents horizontally in rows or vertically in columns
* A gap of 20 pixels is established between elements on a row using the setHGap method

**Determining Event Sources**

* Recall that you must establish a relationship between controls and the event handlers that respond to events
* When appropriate, one event handler object can be used to listen to multiple controls
* The source of the event can be determined by using the getSource method of the event passed to the event handler

**Java Class 13**

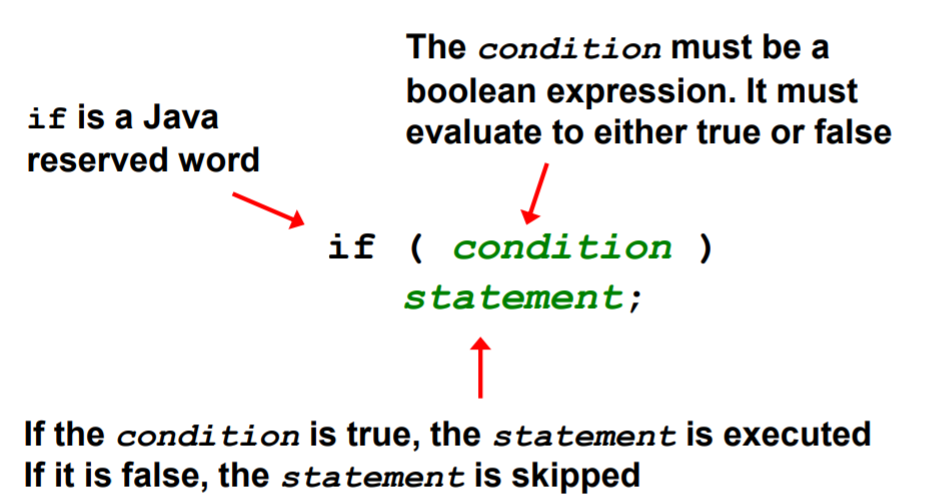
**Conditional Statements**

* A **conditional statement** lets us choose which statement will be executed next
* Conditional statements give us the power to make basic decisions
* The Java conditional statements are the:
  + if and if-else statement
  + switch statement (needed for Project 2)

**Boolean Expressions**

* A condition often uses one of Java's **equality operators** or **relational operators**, which all return boolean results:
* Note the difference between the equality operator (==) and the assignment operator (=)

**The if Statement**

* The if statement has the following syntax:
* 
* An if statement with its boolean condition:
* if (sum > MAX) delta = sum – MAX;
* First, the condition is evaluated; the value of sum is greater than the value of MAX, or it is not
* If the condition is true, the assignment statement is executed; if it isn't, it is skipped

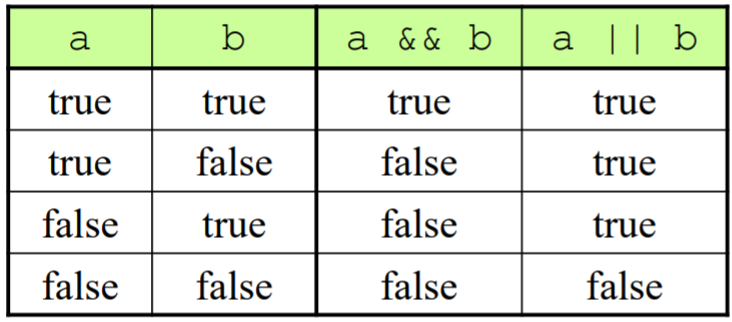
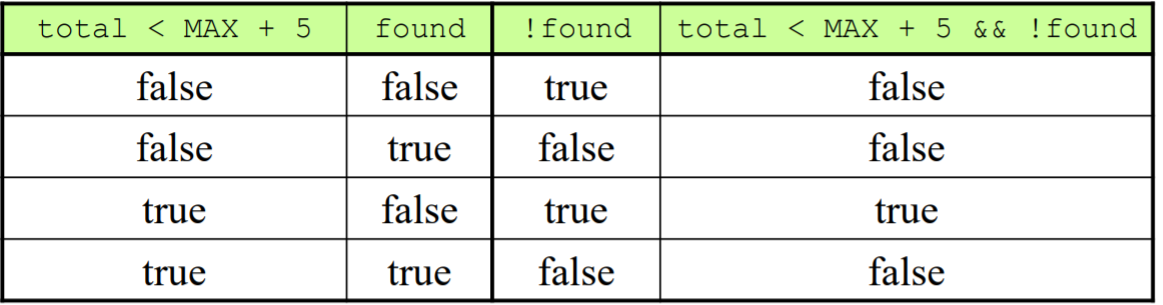
**Logical Operators**

* Boolean expressions can also use the following **logical operators**:
  + ! Logical NOT
  + && Logical AND
  + || Logical OR
* They all take boolean operands and produce boolean results
* Logical NOT is a unary operator (it operates on one operand)
* Logical AND and logical OR are binary operators (each operates on two operands)

**Logical NOT**

* The logical NOT operation is also called logical negation or logical complement
* If some boolean condition a is true, then !a is false; if a is false, then !a is true
* Logical expressions can be shown using a **truth table**:
  + a !a
  + true false
  + false true

**Logical AND and Logical OR**

* A truth table shows all possible true-false combinations of the terms
* Since && and || each have two operands, there are four possible combinations of conditions a and b
* 
* Expressions that use logical operators can form complex conditions
* if (total < MAX+5 && !found)
* System.out.println ("Processing…");
* All logical operators have lower precedence than the relational operators
* The ! operator has higher precedence than && and ||
* 
* **Short-Circuited Operators**
  + The processing of && and || is “short-circuited”
  + If the left operand is sufficient to determine the result, the right operand is not evaluated
  + if (count != 0 && total/count > MAX) System.out.println ("Testing.");
  + If count equals 0, then total/count > MAX is never evaluated
  + This type of processing should be used carefully

**Indentation**

* The statement controlled by the if statement is indented to indicate that relationship
* The use of a consistent indentation style makes a program easier to read and understand
* The compiler ignores indentation, which can lead to errors if the indentation is not correct
* Remember that indentation is for the human reader and is ignored by the compiler
* Despite what the indentation implies, delta will be set to 0 no matter what

**The if-else Statement**

* An **else clause** can be added to an if statement to make an ifelse statement
  + if ( condition )
  + statement1;
  + else
  + statement2;
* If the condition is true, statement1 is executed; if the condition is false, statement2 is executed
* One or the other will be executed, but not both

**Block Statements**

* Several statements can be grouped together into a **block statement** delimited by braces
* A block statement can be used wherever a statement is called for in the Java syntax rules
  + if (total > MAX) { System.out.println ("Error!!"); errorCount++; }
* The if clause, or the else clause, or both, could be governed by block statements
  + if (total > MAX) { System.out.println ("Error!!"); errorCount++; }
  + else { System.out.println ("Total: " + total); current = total\*2; }

**Nested if Statements**

* The statement executed as a result of an if or else clause could be another if statement
* These are called **nested if statements**
* An else clause is matched to the last unmatched if (no matter what the indentation implies)
* Braces can be used to specify the if statement to which an else clause belongs

**Java Class 14**

**Comparing Data**

* When comparing data using boolean expressions, it's important to understand the nuances of certain data types
* Let's examine some key situations:
  + ◦ Comparing floating point values for equality
  + ◦ Comparing characters
  + ◦ Comparing strings (alphabetical order)
  + ◦ Comparing object vs. comparing object references

**Comparing Floating Point Values**

* You should rarely (never) use the equality operator (==) when comparing two floating point values (float or double) because two floating point values are equal only if their underlying binary representations match exactly
* To determine the equality of two floating point numbers, use the following technique:
  + if (Math.abs(f1 - f2) < TOLERANCE) System.out.println ("Essentially equal");
* If the difference between the two floating point values is less than the tolerance, they are considered to be equal
* The tolerance could be set to any appropriate level, such as 0.000001

**Comparing Characters**

* As we've discussed, Java character data is based on the Unicode character set
* Unicode establishes a particular numeric value for each character, and therefore an ordering
* We can use relational operators on character data based on this ordering
* For example, the character '+' is less than the character 'J' because it comes before it in the Unicode character set
* Appendix C provides an overview of Unicode
* In Unicode, the digit characters (0-9) are contiguous and in order
* Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order

**Comparing Strings**

* Remember that in Java a character String is an object
* The equals method can be called with strings to determine if two strings contain exactly the same characters in the same order
* The equals method returns a boolean result
  + if (name1.equals(name2))
    - System.out.println ("Same name");
* We cannot use the relational operators to compare strings because they are not primitives
* The String class contains the compareTo method for determining if one string comes before another
* A call to name1.compareTo(name2)
  + ◦ returns zero if name1 and name2 are equal (contain the same characters)
  + ◦ returns a negative value if name1 is less than name2
  + ◦ returns a positive value if name1 is greater than name2
* Because comparing characters and strings is based on a character set, it is called a **lexicographic ordering**
  + Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed
  + For example, the string "Great" comes before the string "fantastic" because all of the uppercase letters come before all of the lowercase letters in Unicode
  + Also, short strings come before longer strings with the same prefix (lexicographically)
  + Therefore "book" comes before "bookcase"

**Comparing Object References**

* To see if two object references are the same (which means they refer to the same object on the heap), use the equality operator (==)
  + Foo a = new Foo();
  + Foo b = new Foo();
  + Foo c = a;
  + if (a == b) false
  + if (a == c) true
  + if (b == c) false

**Repetition Statements**

* **Repetition statements** allow us to execute a statement multiple times
* Often they are referred to as **loops**
* Like conditional statements, they are controlled by boolean expressions
* Java has three kinds of repetition statements: while, do, and for loops

**The while Statement**

* A **while statement** has the following syntax:
  + while ( **condition** )
    - **statement**;
* If the condition is true, the statement is executed
* Then the condition is evaluated again, and if it is still true, the statement is executed again
* The statement is executed repeatedly until the condition becomes false
* An example of a while statement:
* If the condition of a while loop is false initially, the statement is never executed
* Therefore, the body of a while loop will execute zero or more times
* **Sentinel Values**
  + Let's look at some examples of loop processing
  + A loop can be used to maintain a running sum
  + A **sentinel value** is a special input value that represents the end of input
* **Input Validation**
  + A loop can also be used for input validation, making a program more robust
  + It's generally a good idea to verify that input is valid (in whatever sense) when possible
* **Infinite Loops**
  + The body of a while loop eventually must make the condition false
  + If not, it is called an **infinite loop**, which will execute until the user interrupts the program
  + This is a common logical error
  + You should always double check the logic of a program to ensure that your loops will terminate normally
  + This loop will continue executing until interrupted (Control-C) or until an underflow error occurs

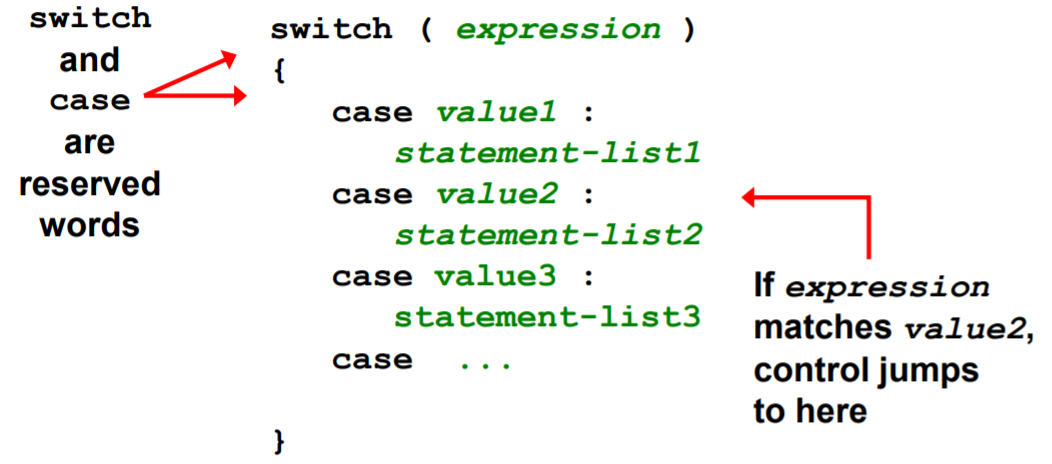
**The ArrayList Class**

* An ArrayList object stores a list of objects, and is often processed using a loop
* The ArrayList class is part of the java.util package
* You can reference each object in the list using a numeric index
* An ArrayList object grows and shrinks as needed, adjusting its capacity as necessary
* Index values of an ArrayList begin at 0 (not 1):
  + 0 "Bashful"
  + 1 "Sleepy"
  + 2 "Happy"
  + 3 "Dopey"
  + 4 "Doc"
* Elements can be inserted and removed
* The indexes of the elements adjust accordingly
* Some ArrayList methods:
  + boolean add(E obj)
  + void add(int index, E obj)
  + Object remove(int index)
  + Object get(int index)
  + boolean isEmpty()
  + int size()
* The type of object stored in the list is established when the ArrayList object is created:
  + ArrayList<String> names = new ArrayList<String>();
  + ArrayList<Book> list = new ArrayList<Book>();
* This makes use of Java generics, which provide additional type checking at compile time
* An ArrayList object cannot store primitive types, but that's what wrapper classes are for

**Java Class 15**

Java: an island in Indonesia to the south of Borneo; one of the world's most densely populated regions in Asia made up of an archipelago including more than 13,000 islands; achieved independence from the Netherlands in 1945; the principal oil producer in the Far East and Pacific regions; the capital of Indonesia, Jakarta is located on the island of Java.

**The switch Statement**

* The general syntax of a switch statement is:
* 
* Often a break statement is used as the last statement in each case's statement list
* A *break* statement causes control to transfer to the end of the switch statement
* If a break statement is not used, the flow of control will continue into the next case
* Sometimes this may be appropriate, but often we want to execute only the statements associated with one case
* A switch statement can have an optional *default* case
* The default case has no associated value and simply uses the reserved word default
* If the default case is present, control will transfer to it if no other case value matches
* If there is no default case, and no other value matches, control falls through to the statement after the switch
* The type of a switch expression must be integers, characters, enumerated types, or, as of Java 7, strings
* You cannot use a switch with floating point values
* The implicit boolean condition in a switch statement is equality
* You cannot perform relational checks with a switch statement

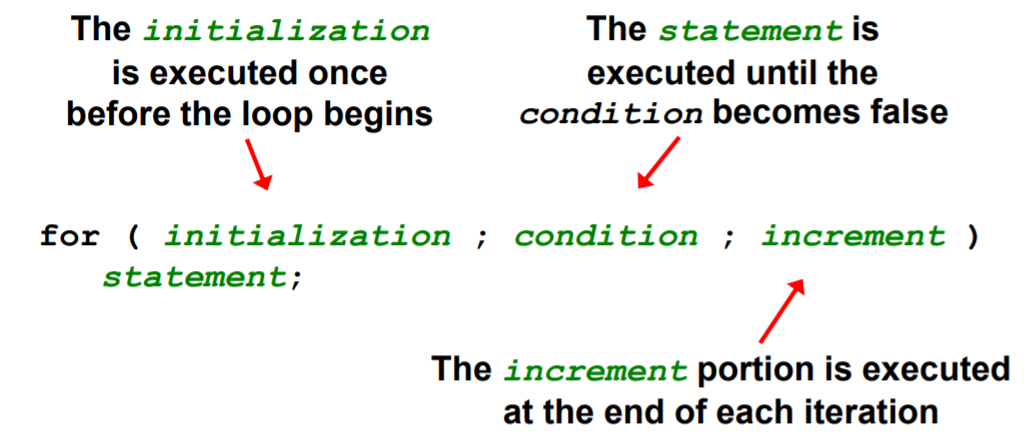
**The Conditional Operator**

* The conditional operator evaluates to one of two expressions based on a boolean condition
* Its syntax is:
  + **condition** ? **expression1** : **expression2**;
* If the **condition** is true, **expression1** is evaluated; if it is false, **expression2** is evaluated
* The conditional operator is similar to an if-else statement, except that it is an expression that returns a value
* For example:
  + larger = ((num1 > num2) ? num1 : num2);
* If num1 is greater than num2, then num1 is assigned to larger; otherwise, num2 is assigned to larger
* The conditional operator is *ternary* because it requires three operands
* Another example:
  + System.out.println ("Your change is " + count + ((count == 1) ? "Dime" : "Dimes"));
* If count equals 1, the "Dime" is printed
* If count is anything other than 1, then "Dimes" is printed

**The do Statement**

* A do statement has the following syntax:
  + do { **statement-list**; } while (**condition**);
* The **statement-list** is executed once initially, and then the **condition** is evaluated
* The statement is executed repeatedly until the condition becomes false
* An example of a do loop:
* The body of a do loop executes at least once
  + int count = 0;
  + do { count++; System.out.println (count); while (count < 5);}

**The for Statement**

* A for statement has the following syntax:
* 
* A for loop is functionally equivalent to the following while loop structure:
  + **initialization**; while ( **condition** ) { **statement**; **increment**; }
* An example of a for loop:
  + for (int count=1; count <= 5; count++)
    - System.out.println (count);
* The initialization section can be used to declare a variable
* Like a while loop, the condition of a for loop is tested prior to executing the loop body
* Therefore, the body of a for loop will execute zero or more times
* The increment section can perform any calculation:
  + for (int num = 100; num > 0; num -= 5) System.out.println (num);
* A for loop is well suited for executing statements a specific number of times that can be calculated or determined in advance

Conditionals and loops enhance our ability to generate interesting graphics

**Java Class 16**

**Managing Fonts**

* The Font class represents a character font, which specify what characters look like when displayed
* A font can be applied to a Text object or any control that displays text (such as a Button or Label)
* A font is specifies:
  + ◦ font family (Arial, Courier, Helvetica)
  + ◦ font size (in units called points)
  + ◦ font weight (boldness)
  + ◦ font posture (italic or normal)
* A Font object is created using either the Font constructor or by calling the static font method
* The Font constructor can only take a font size, or a font family and size
* To set the font weight or font posture, use the font method, which can specify various combinations of font characteristics
* Note that setting the text color is not a function of the font applied
* It's set through the Text object directly
* The same is true for underlined text (or a "strike through" effect)

**Check Boxes**

* A check box is a button that can be toggled on or off
* It is represented by the JavaFX CheckBox class
* Checking or unchecking a check box produces an action event

**Radio Buttons**

* Let's look at a similar example that uses radio buttons
* A group of radio buttons represents a set of mutually exclusive options – only one button can be selected at any given time
* When a radio button from a group is selected, the button that is currently "on" in the group is automatically toggled off
* To establish a set of mutually exclusive options, the radio buttons that work together as a group are added to a ToggleGroup object
* The setToggleGroup method is used to specify which toggle group a button belongs to
* The isSelected method of a radio button returns true if that button is currently "on

**Graphic Transformations**

* A JavaFX transformation changes the way a node is presented visually
  + ◦ translation – shifts the position along the x or y axis
  + ◦ scaling – causes the node to appear larger or smaller
  + ◦ rotation – rotates the node around its center point
  + ◦ shearing – rotates one axis so that the x and y axes are no longer perpendicular

**Translation**

* The following creates two rectangles in the same position, then shifts the second one:
  + Rectangle rec1 = new Rectangle(100, 100, 200, 50);
  + rec1.setFill(Color.STEELBLUE);
  + Rectangle rec2 = new Rectangle(100, 100, 200, 50);
  + rec2.setFill(Color.ORANGE);
  + rec2.setTranslateX(70);
  + rec2.setTranslateY(10);

**Scaling**

* The following displays two ImageView objects, the second scaled to 70%:
  + Image img = new Image("water lily.jpg");
  + ImageView imgView1 = new ImageView(img);
  + ImageView imgView2 = new ImageView(img);
  + imgView2.setX(300);
  + imgView2.setScaleX(0.7);
  + imgView2.setScaleY(0.7);

**Rotation**

* The parameter to setRotate determines how many degrees the node is rotated
* If the parameter positive, the node is rotated clockwise
* If the parameter is negative, the node is rotated counterclockwise
  + Rectangle rec = new Rectangle(50, 100, 200, 50);
  + rec.setFill(Color.STEELBLUE);
  + rec.setRotate(40);
  + Text text = new Text(270, 125, "Tilted Text!");
  + text.setFont(new Font("Courier", 24));
  + text.setRotate(-15);
* To rotate a node around a point other than its center point, create a Rotate object and add it to the node's list of transformations
* The following rotates a node 45 degrees around the point (70, 150):
  + node.getTransforms().add(new Rotate(45, 70, 150));

**Shearing**

* Shearing is accomplished by creating a Shear object and adding it to this list of transformations
* The following applies a shear of 40% on the x axis and 20% on the y axis to an ImageView object:
  + Image img = new Image("duck.jpg");
  + ImageView imgView = new ImageView(img);
  + imgView.getTransforms().add(new Shear(0.4, 0.2));

**Transformations on Groups**

* Transformations can be applied to any JavaFX nodes
  + ◦ shapes, images, controls
  + ◦ groups and panes
* When applied to a group or pane, the transformation is applied to each node it contains

**Java Class 17**

**LA Air Traffic Control**

On September 14th, 2004, the LA air traffic control center suddenly lost voice control with 400 planes that it was tracking. The backup system crashed a minute after it was activated. Officially the incident was blamed on human error, but the root cause was traced to a software problem that had to do with an internal timer that determined when tests would be run. The timer was allowed to run down to zero, which then shut the system down to run the tests.

**GUI Design**

* We must remember that the goal of software is to help the user solve the problem
* To that end, the GUI designer should:
  + ◦ Know the user
  + ◦ Prevent user errors
  + ◦ Optimize user abilities
  + ◦ Be consistent
* Each of these will be looked out in more detail

**Know the User**

* Knowing the user implies an understanding of:
  + ◦ the user's true needs
  + ◦ the user's common activities
  + ◦ the user's level of expertise in the problem domain and in computer processing
* We should also realize these issues may differ for different users
* Remember, to the user, the interface is the program

**Prevent User Errors**

* Whenever possible, we should design user interfaces that minimize possible user mistakes
* We should choose the best GUI components for each task
* For example, in a situation where there are only a few valid options, using a menu or radio buttons would be better than an open text field
* Error messages should guide the user appropriately

**Optimize User Abilities**

* Not all users are alike – some may be more familiar with the system than others
* Knowledgeable users are sometimes called power users
* We should provide multiple ways to accomplish a task whenever reasonable
  + ◦ "wizards" to walk a user through a process
  + ◦ short cuts for power users
* Help facilities should be available but not intrusive

**Be Consistent**

* Consistency is important – users get used to things appearing and working in certain ways
* Colors should be used consistently to indicate similar types of information or processing
* Screen layout should be consistent from one part of a system to another
* For example, error messages should appear in consistent locations

**Mouse Events**

* JavaFX nodes can generate several types of mouse-based events:
* The MouseEvent object representing the event can be used to obtain the mouse position
* There are convenience methods for setting the handler for each type of mouse event (such as setOnMousePressed)
* A stream of mouse moved or mouse dragged events occur while the mouse is in motion
* This essentially allows the program to track the movement in real time
* Using the mouse to "draw" a shape into place is called rubberbanding

**Key Events**

* There are three JavaFX events related to the user typing at the keyboard:
* The getCode method of the event object returns a code that represents the key that was pressed

**Quiz #2 Review**

Java methods can only return primitive types. – F

* Java methods can also return objects such as String.

Formal parameters are those that appear in the method call and actual parameters are those that appear in the method header – F

* The question has the two definitions reversed. Formal parameters are those that appear in the method header, actual parameters are the parameters in the method call (those being passed to the method).

All Java classes must contain a main method, which is the first method executed when the Java class is called. – F

* Only the driver program requires a main method. The driver program is the one that is first executed in any Java program, but it may call upon other classes as needed, and these other classes do not need main methods.

Java methods can return more than one item if they are modified with the reserved word continue, as in public continue int foo() {...} – F

* All Java methods return a single item, whether it is a primitive data type an object, or void. The reserved word continue is used to exit the remainder of a loop and test the condition again.

The following method header definition will result in a syntax error: public void aMethod(); - T

* The reason for the syntax error is because it ends with a ; symbol. It instead needs to be followed by {} with 0 or more instructions inside of the brackets. An abstract method will end with a ; but this header does not define an abstract method.

A method defined in a class can access the class's instance data without needing to pass them as parameters or declare them as local variables. – T

* The instance data are globally available to all of the class's methods and therefore the methods do not need to receive them as parameters or declare them locally. If variables of the same name as instance data were declared locally inside a method then the instance data would be "hidden" in that method because the references would be to the local variables.

Defining formal parameters requires including each parameter's type. – T

* In order for the compiler to check to see if a method call is correct, the compiler needs to know the types for the parameters being passed. Therefore, all formal parameters (those defined in the method header) must include their type. This is one element that makes Java a strongly typed language.

Every class definition that is used to create objects must include a constructor. – F

* Java allows classes to be defined without constructors. However, there is a default constructor that is used in such a case.

While multiple objects of the same class can exist, in a given program there can only be one version of each class. – T

* A class is an abstraction; that is, it exists as a definition, but not as a physical instance. Physical instances are created when an object is instantiated using new. Therefore, there can be many objects of type String, but only one String class.

An object should be encapsulated in order to guard its data and methods from inappropriate access – T

* Encapsulation is the concept that objects should be protected from accidental (or purposeful) misuse.

Accessors and mutators provide mechanisms for controlled access to a well-encapsulated class. – T

* Accessors provide read access to variables that otherwise would be inaccessible. Mutators provide write access to otherwise inaccessible variables.

A GUI control sets up an event, but it is the programmer who writes the code for the event handler, which executes when an event occurs. – T

Because an Image cannot directly be added to a container, it must be displayed using an ImageView object. – T

In Java, selection statements consist only of if and if-else statements – F

* This list omits switch statements.

In Java, the symbol "=" and the symbol "==" are used synonymously (interchangeably). – F

* "=" is used for assignment statements while "==" is used to test equality.

The statement {} is a legal block – T

* A block consists of {, followed by zero or more Java statements, followed by }. So it is acceptable to have no statements between the brackets. Situations where this is necessary occur in Java, particularly when implementing methods of abstract classes, something you will study later.

The statement: if (a >= b) a++; else b--; will do the same thing as the statement: if (a < b) b--; else a++;. – T

* We can reverse the if clause and else clause if we reverse the condition. The opposite condition of (a >= b) is (a < b) so this works out logically. Note that if we used the condition (a <= b) then the resulting statement would not do the same thing as the original if (a >= b).

An if statement may or may not have an else clause, but an else clause must be part of an if statement. – T

* Java allows for either if or if-else statements. But else is only used as part of an if statement.

In order to compare int, float and double variables, you can use , ==, !=, <=, >=, but to compare char and String variables, you must use compareTo(), equals() and equalsIgnoreCase(). – F

* You can also directly compare char variables using , ==, !=, <=, >=, but you must use compareTo(), equals() and equalsIgnoreCase() for any String comparisons.

Assume that boolean done = false, int x = 10, and int y = 11. Then the expression (!done && x <= y) is true. – T

* Since done is false, !done is true. Since 10 < 11, x <= y is true. Therefore, the entire expression is true.

Assume that boolean done = false, int x = 10, int y = 11, String s = "Help" and String t = "Goodbye". Then the expression (s.concat(t).length() < y) is true. – F

* Concatenating s and t gives a String that is 11 characters long and 11 < 11 is fals

Assume that boolean done = false, int x = 10, int y = 11, String s = "Help" and String t = "Goodbye". Then the expression (done || s.compareTo(t) < 0) is true.- F

* Both done is false and s.compareTo(t) < 0 is false since s does not come before t alphabetically, so the entire expression is false.

A switch statement must have a default clause. – F

* The default clause is optional.

Each switch case statement must terminate with a break statement. – F

* They often do but if the break statement is not present, the flow of control continues into the next case.

Control in a switch statement jumps to the first matching case. – T

* The switch expression is evaluated and control jumps to the first matching case, then continues from there

The following for loop is an infinite loop:: for(int j = 0; j < 1000;) i++; - T

* This loop initializes j to 0 and compares it to 1000, but does not alter j after each loop iteration. In reality, the loop will terminate with a run-time error eventually once i becomes too large to store in memory, but logically, this is an infinite loop.

It is possible to convert any type of loop (while, do, for) into any of the other two Java loops. – T

* All loop statements have equivalent expressive power.

The following loop is syntactically valid: for(int j = 0; j < 1000; j++) j--; - T

* The Java compiler does not care that you are incrementing j in the loop but decrementing j in the loop body. Logically, this loop makes no sense because j will continuously be incremented and decremented so that it never reaches 1000, but there is nothing wrong with the loop syntactically.

In Java, it is possible to create an infinite loop out of while and do loops but not for loops. - F

* It is true that while and do loops can be infinite loops, but it is also true that Java for loops can be infinite loops. This is not true in some other programming languages where for loops have a set starting and ending point, but Java for loops are far more flexible than most other language's for loops.

A conditional operator is virtually the same as a switch statement. – F

* The conditional operator is more like an if-else statement.

A for statement is normally used when you do not know how many times the loop should be executed. – F

* A for statement is normally used when you do know how many times the loop should be executed.

A loop can be used in a GUI to draw concentric circles – T

The code below presumably used to create Box objects is syntactically correct. – T

* public class Box {

double length;

double width;

double height;

Box(double l, double w, double h)

{ length = l; width = w; height = h; }

double volume() { return length \* width \* height; } }

Write the statement to instantiate a Box object, blueBox, with a length of 6, height of 2, and width of 4.

* Box blueBox = new Box(6,4,2);

The following set of statements will add 1 to x if x is positive and subtract 1 from x if x is negative but leave x alone if x is 0. if (x > 0) x++; else x--; - F

* x- is done if x is not positive, thus if x is 0, x becomes -1 which is the wrong answer.

Assume that count is 0, total is 20, and max is 1. When the following code is executed, the condition short circuits and the assignment statement is not executed. if (count != 0 || total / count > max) max = total / count; - F

* The above statement would be true if the || was replaced with &&.

Find the error in the following code and fix the code so that this statement is a legal if-else statement. if (x < 0); x++; else x--;

* if (x < 0) x++; else x--;

Rewrite the following if-else statement using a conditional operator. if (x > y) z = x; else z = y;

* z = (x > y) ? x : y;

A do loop should be used in a situation where you want to do data verification by asking the user for a value, and only if the user has entered an improper value does your code repeat and try again. – T

* The do loop is used if you want to execute the loop body at least one time. This is useful for data verification (asking the user for a value, and only if the user has entered an improper value does your code repeat and try again).

The following JavaFX statement will rotate an Ellipse named lipse 30 degrees in the counterclockwise direction. lipse.setRotate(30); - F

* This statement will rotate lipse 30 degrees in the clockwise direction.