**More Examples**

In the previous lecture, I mentioned that in addition to using a boolean variable, a conditional statement may use a boolean literal, boolean operator, or a boolean return value from a method.  Here are some examples of these other forms:

// boolean literal

if( true ) {

}

// boolean operator (>= is 'greater than or equal to')

if( myAge >= 40 ) {

}

// boolean return value from a method

public boolean doSomething() {

return true;

}

if( doSomething() ) {

}

**Variable Scope**

Any variable that is declared in the conditional statement is local to that block. For example, the following would be illegal:

int age = 40;

if (age >= 25) {

// Note the "String" reference type.

// That means we're declaring the variable

// in this scope.

String message = "25 years or older.";

}

// ILLEGAL, message is out of scope

System.out.println(message);

If the variable is assigned inside the conditional, but is also to be used outside of it, make sure the variable is declared outside as well.

String message = null; // Declaration of "message"

int age = 40;

if (age >= 25) {

// Note that we've dropped the "String"

// reference type

// Which means we're referring to, rather than

// declaring the message variable

message = "25 years or older.";

}

System.out.println(message); // LEGAL

**Braces**

I mentioned that if a conditional doesn't use braces, only the first statement is considered a part of the conditional block (any subsequent statements are just normal statements, which will be executed every time).

For example, due to the lack of braces, line A  below will be executed every time, even when the if block is evaluated to be false.

if (age >= 25)

message = "25 years or older.";

otherMessage = "More text..."; // LINE A

... and the same goes for this example... even when age >= 25 (so the else block isn't entered), Line A is printed.

if (age >= 25)

message = "25 years or older.";

else

message = "Younger than 25.";

otherMessage = "More text..."; // LINE A

It should also be noted that when there is an else-if and/or else block, having multiple statements, without braces, in anything but the last block is illegal.

For example, the following example is illegal:

if (age >= 25)

message = "25 years or older.";

otherMessage = "More text..."; // ILLEGAL

else if (age >= 20)

message = "Between 20 years and 24 years old";

otherMessage = "More text..."; // ILLEGAL

else

message = "Younger than 20.";

otherMessage = "More text..."; // LEGAL

**Bottom line... USE YOUR BRACES!**

**Chapter 7 – “Conditionals – Switch”**

**Break vs. Return**

In the "Introduction to Objects" section, I mentioned that it's OK to declare an empty return statement when a method returns void.  As a reminder, the following is odd (not really something you'd see in the "real world"), but legal:

public void doSomething() {

return;

}

Now that you've been introduced to conditional statements, you can see some "real world" example of how this might be used.  In these examples, if a condition is true, we don't want to do any additional processing so we return out of the method.  This is different than "break," which would only cause us to break out of a loop or switch statement (rather than the entire method). I will say, however, I'm not a big fan of having return statements in the middle of a method... Having multiple exit points can lead to hard to read, if not buggy, code.

In any case, here are some legal examples of returning when the method's return type is void.

public void doSomething() {

if( isTrue ) {

return; // Exits the entire method

} else {

// Do some work...

}

// Do some work if we didn't already

// return out of the method...

}

and...

public void doSomething() {

switch (someVariable) {

case 0: {

return; // exits the entire method

}

case 1: {

// Do some work...

break; // exits the switch

}

case 2: {

// Do some work...

break; // exits the switch

}

}

// If we didn't return out of the method

// do some more work...

}

**Variable Scope**

Just like if / else if / else statements, any variables declared inside of a case are local to that code block.

switch (someVariable) {

case 0: {

// Declaration (rather than reference)

String msg = "Hello!";

break;

}

}

System.out.println(msg); // ILLEGAL

So make sure any variables you want to reference outside of the switch were also declared outside of the switch:

String msg = null; // Declaration

switch (someVariable) {

case 0: {

msg = "Hello!"; // Reference

break;

}

}

System.out.println(msg); // Legal

**Switch Content**

One final note: the only legal members of a switch statement are "case" and "default" statements.  For example, you can't have variables declared:

switch (someVariable) {

String msg = "Hello"; // ILLEGAL

case 0: {

msg = "Hello!";

break;

}

default: {

msg = "Hey!";

break;

}

}

**Chapter 7 – “Operators Pt 2” Lecture**

**Boolean Operators**

These operators, which return true or false, were mentioned in the Operators Part 1 lecture, but let's go over a few more examples.  To start, each of the examples will use the following variables:

boolean courseInSession = true;

int lectureCount = 150;

! is the "logical complement" operator.  It allows us to test for a false value:

if( !courseInSession ) {

System.out.println("Course is not in session.");

}

 ... or it can flip a true value to false, and a false value to true:

courseInSession = !courseInSession;

!= is the "not equal to" operator:

if( lectureCount != 150 ) {

System.out.println("The lecture count is something other than 150");

}

The == operator checks for equality:

if( lectureCount == 150 ) {

System.out.println("The lecture count is 150");

}

And the rest check to see if a value is less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=):

if( lectureCount < 10 ) {

System.out.println("Lecture count is less than 10");

} else if( lectureCount >= 20 && lectureCount <= 30 ) {

System.out.println("Lecture count is between 20 and 30 [inclusive]");

} else if( lectureCount > 100 ) {

System.out.println("Lecture count is 101 or greater.");

}

**Arithmetic Operators**

+ is add (or concatenation when used with a String), - is subtract, \* is multiply, and / is divide. The use of those operators should be obvious. The one arithmetic operator that deserves further discussion is the modulo or remainder operator: %.  This operator returns the remainder of a division operation.  For example:

10 % 2 == 0 (10 / 2 == 5 with a remainder of 0)

10 % 4 == 2 (10 / 4 == 2 with a remainder of 2)

etc.

Assignment Operators

= is an assignment operator.  It can be used on its own (int x = 30;) or it can be used with arithmetic operations as well:

x = x + 5;

is the same thing as:

x += 5;

Other examples:

x -= 5; // Same as x = x - 5;

x \*= 5; // Same as x = x \* 5;

X /= 5; // Same as x = x / 5;

x %= 5; // Same as x = x % 5;

**Chapter 8 – “Immutable Strings” Chapter**

We've already seen the concatenation operator (+) and how it's used with Strings.  But what happens when it's used with numeric primitives?

The rule is pretty simple: the + is treated as an addition operator until it is being applied to a String.  From that point on, the + is treated as a String concatenation operator.

For Example:

int employeeId = 2812;

int deptId = 98;

System.out.println(employeeId + deptId); // Prints 2910

System.out.println(employeeId + ":" + deptId); // Prints 2812:98

System.out.println(employeeId + deptId + ":" + employeeId + deptId);

// The line above prints 2910:281298

**Chapter 10 – Chaining Constructors**

I mentioned that when chaining constructors, the call to the super constructor must be the first statement in the constructor.  And that's true.  Assuming the Person class contains a no-arg constructor, the following constructor is valid:

public Employee() {

super();

deptId = 281;

}

...whereas this one is not:

public Employee() {

deptId = 281;

super();

}

I want to clarify that using super with the dot operator, to invoke a super type's method (other than a constructor) or access its state, is NOT bound by the first statement rule.

Both of these examples are valid:

public void driveLikeDad() {

super.drive();

doSomethingElse();

}

and...

public void driveLikeDad() {

doSomethingElse();

super.drive();

}

**Chapter 10 – “The Three Faces of Final” Lecture**

**Switch Statements Revisited**

When we discussed Switch statements in the Java Syntax section, I mentioned that each case statement had to use a value that was understood at compile time (variables and return values from methods cannot be used).  At that point in the course, the only value that fit that description was a literal value.  For example, the following code uses char literals for each case statement:

char studentGrade = 'A';

switch (studentGrade) {

case 'A':

System.out.println("Student received an A");

break;

case 'B':

System.out.println("Student received a B");

break;

default:

System.out.println("Student received a C or lower");

}

Now that we've learned about "final," we can add constants to our short list of valid case values.  Constants works because their value is known at compile time.  So the following is legal:

final char A\_GRADE = 'A';

final char B\_GRADE = 'B';

char studentGrade = 'A';

switch (studentGrade) {

case A\_GRADE:

System.out.println("Student received an A");

break;

case B\_GRADE:

System.out.println("Student received a B");

break;

default:

System.out.println("Student received a C or lower");

}

**Know This For The Exam: Class/Object Invocation Order**

When an object in an inheritance chain is instantiated for the first time, this is the order that the code is executed:

1. All of the static variables are defined in the base class (with default values).  For example, the code public static String firstName = "Jason";  would result in the creation of the variable firstName  but in this step, it is assigned the default value of null (rather than the explicit value of "Jason").
2. All of the static initialization blocks and explicit values assigned to static variables (such as "Jason" above), are executed in the base class, in the order they are written (from top to bottom).
3. All of the static variables are defined for the immediate child of the base class (with default values).
4. All of the static initialization blocks and explicit values assigned to static variables are executed for the immediate child of the base class, in the order they are written (from top to bottom).
5. Steps 3 - 4 are repeated all the way down the object hierarchy until the instantiated object type is reached.
6. All of the instance variables for the base class are defined (with default values).
7. All of the instance initialization blocks and explicit values assigned to instance variables are executed for the base class, in the order they are written (from top to bottom).
8. The constructor for the base class is executed.
9. All of the instance variables for the immediate child of the base class are defined (with default values).
10. All of the instance initialization blocks and explicit values assigned to instance variables are executed for the immediate child of the base class, in the order they are written (from top to bottom),
11. The constructor for the immediate child of the base class is executed.
12. Steps 9 - 11 are repeated all the way down the object hierarchy until the instantiated object type is reached.

Note that the first steps that execute "static" code will only happen once during the entire life of the class, and may have been executed before an object has been instantiated (such as when a reference variable is created or a static member is accessed).

Here's an example to illustrate the order class and object instantiation is executed. The following classes are defined in three separate files (and note that their members are organized in a haphazard manner just to make sure you're paying attention!):

**Person**

public class Person {

{

System.out.println("Person: First Instance Initialization Block");

}

static {

System.out.println("Person: First Static Block");

}

{

System.out.println("Person: Second Instance Initialization Block");

}

static {

System.out.println("Person: Second Static Block");

}

public Person() {

System.out.println("Person()");

}

public void sayHello() {

System.out.println("Person: Hello!");

}

}

**Employee**

public class Employee extends Person {

{

System.out.println("Employee: First Instance Initialization Block");

}

static {

System.out.println("Employee: First Static Block");

}

{

System.out.println("Employee: Second Instance Initialization Block");

}

static {

System.out.println("Employee: Second Static Block");

}

public Employee() {

System.out.println("Employee()");

}

}

**Instructor**

public class Instructor extends Employee {

{

System.out.println("Instructor: First Instance Initialization Block");

str1 = "First Instance Initialization String";

}

static {

System.out.println("Instructor: First Static Block");

str1 = "First Static Initialization String";

}

public static String str1 = "Explicit Initialization String";

{

System.out.println("Instructor: Second Instance Initialization Block");

str1 = "Second Instance Initialization String";

}

static {

System.out.println("Instructor: Second Static Block");

str1 = "Second Static Initialization String";

}

public Instructor() {

System.out.println("Instructor()");

}

}

The following code...

Instructor i = new Instructor();

i.sayHello();

System.out.println(Instructor.str1);

...would generate this output :

(assuming the Person, Employee, and Instructor classes have not been previously referenced, otherwise the static blocks wouldn't be executed again)

Person: First Static Block

Person: Second Static Block

Employee: First Static Block

Employee: Second Static Block

Instructor: First Static Block

Instructor: Second Static Block

Person: First Instance Initialization Block

Person: Second Instance Initialization Block

Person()

Employee: First Instance Initialization Block

Employee: Second Instance Initialization Block

Employee()

Instructor: First Instance Initialization Block

Instructor: Second Instance Initialization Block

Instructor()

Person: Hello!

Second Instance Initialization String

NOTE: If another Instructor object is instantiated, the static blocks would be skipped, since that only happens once during the life of the class.

**Chapter 11 – “Polymorphism: Part 2” Lecture**

Throughout this section, I've mentioned that a reference variable may only send messages that are available to its type.  In other words imagine that a Person class has a setFirstName method, and an Employee class has a setSalary method:

Person p = new Employee();

// Legal since setFirstName is available to Person

p.setFirstName("Jane");

// Illegal since salary is available

// to an Employee, not a Person.

p.setSalary(80\_000);

I want to be clear, however, that "being available to an object" is not the same as "being declared inside an object."  Let's say we have a subtype of Employee called Instructor.  If we create an Employee reference variable, we can call any member available to Employee - including those members it inherits.  For example:

Employee e = new Instructor();

e.setFirstName("Jane");

... is legal because Employee inherits the setFirstName method from Person.  It wasn't declared inside Employee, but it was available to Employee.  So that's why I specifically say that a reference variable is limited to the members that are available to, rather than defined inside, the class itself.

**Chapter 12 – “Abstract Classes: Part 2” Lecture**

**Modifiers**

When a method is described as being abstract, it doesn't have an implementation.  It relies on a subtype to implement the method.  Therefore, certain modifiers, such as private, static, and finalcannot be used on an abstract method.  If you think about this logically, it makes sense.  Private, static, and/or final methods cannot be overridden, and all abstract methods must be overridden.  Therefore these modifiers cannot appear in an abstract method declaration.  By the way, that's the trick to remembering what's legal and not legal when defining a concrete implementation of an abstract method: they follow the same rules declared for overriding methods.

**Inheritance**

When we say that a concrete subtype must make sure all abstract methods are overridden, it doesn't necessarily mean that all of the abstract methods are implemented in the subtype.  Basically, the rule is that for a class to be concrete, it must make sure that any method call has some concrete definition that can be invoked.  And that concrete definition could be defined in a supertype (even in an abstract class!)

For example, imagine there are three classes: Person (abstract), Employee (abstract), and Consultant (concrete).  For Consultant to compile, all of the abstract methods in Person and Employee must have a concrete implementation.  As such the following compiles:

public abstract class Person {

protected String firstName;

public abstract String getName();

}

public abstract class Employee extends Person {

protected String title;

public abstract double pay();

public String getName() {

return title + ": " + firstName;

}

}

public class Consultant extends Employee {

public double pay() {

return 80\_000.0;

}

}

Notice that Person defines an abstract method (getName()), and it is not implemented in the Consultant class. This is perfectly OK because Employee defines getName(). When we instantiate a Consultant and call getName(), the concrete implementation defined in Employee is available... and that satisfies the main rule: a class may become concrete (non-abstract), as long as all abstract methods in its hierarchy have a corresponding concrete implementation.

**Chapter 12 – “Interfaces: Part 3”**

While an IDE or a compiler will complain if you make this mistake, you won't have these tools when taking the exam.  So remember: all interface methods are public, even if they don't use the public keyword.

Therefore given the following legal interface:

public interface Payable {

double pay();

}

Unlike interface methods, concrete implementations MUST include the public modifier.  Therefore, the following code will not compile because pay() is missing the "public" modifier.

public class Consultant implements Payable {

double pay() {

return 80\_000.0;

}

}

The correct code is:

public class Consultant implements Payable {

public double pay() {

return 80\_000.0;

}

}

**Chapter 15 – “Multi-Dimensional Arrays” Lecture**

**Balance the Brackets**

Make sure both sides are balanced with the same number of square brackets:

// legal

int[][] credentials = new int[3][2];

/\* Illegal. Missing the 2nd dimension \*/

int[][] credentials2 = new int[3];

**Square Bracket Placement**

Just like regular arrays, the square brackets can go to the right of the type, or the right of the identifier:

int[][] credentials = new int[3][2];

or

int credentials[][] = new int[3][2];

They can be split up as well (though this is uncommon). For example a 3 dimensional array could legally be declared as:

int[] moreStuff[][] = new int[3][3][2];

or

int[][] moreStuff[] = new int[3][3][2];

**Initializing the Final Dimension**

All dimensions must be given a size during its definition, except for the final dimension.  It is legal to initialize the final dimension after it has been defined.  For example the code below is legal.

int[][][] moreStuff = new int[3][3][];

moreStuff[0][2] = new int[2]; // Line A

moreStuff[0][2][0] = 0;

moreStuff[0][2][1] = 1;

While this is legal, note that Line A is adding a 3rd dimension to only one member of 2nd dimension.  In other words, if you tried to access another 2nd dimension value and add an int to its 3rd dimension, it would throw a NullPointerException at runtime:

/\* ILLEGAL! Only [0][2] has a third dimension. \*/

moreStuff[0][1][0] = 0;

And don't forget, only the final dimension can have its size defined in a separate statement.  The following three lines of code are illegal and would not compile:

int[][][] moreStuff = new int[3][][3];

int[][][] moreStuff2 = new int[][3][3];

int[][][] moreStuff3 = new int[][][3];

**Chapter 17 – “Limitations of Collections” Lecture**

**Revisiting Overloading Methods: Order of Precedence**

Now that we've learned about wrapper classes, we can discuss one more aspect about overloading methods: how a method is selected when multiple options could potentially work.  Specifically, when invoking a method with a primitive argument, it will use the following order of precedence when trying to find a matching method:

1. The exact primitive type of the argument
2. If the argument is a byte, short, char, int, or long:
   1. The next larger size whole number primitive available (byte, short, int, and long)
   2. The smallest size decimal primitive available (float, double)
3. If the argument is a float:
   1. A double primitive
4. A wrapper class type (has to be an exact match)
5. Varargs (exact primitive type)
6. If the argument is a byte, short, char, int, or long:
   1. The next larger size whole number varargs available (byte, short, int, and long)
   2. The smallest size decimal varargs available (float, double)
7. If the argument is a float:
   1. A double varargs
8. Varargs (exact Wrapper class)

For Example:

public class MethodPicker {

// Method A

public void pickMe(int x) {

System.out.println ("int");

}

// Method B

public void pickMe(long x) {

System.out.println ("long");

}

// Method C

public void pickMe(Integer x) {

System.out.println ("Integer");

}

// Method D

public void pickMe(int... x) {

System.out.println ("int...");

}

// Method E

public void pickMe(long... x) {

System.out.println ("long...");

}

public static void main(String[] args) {

MethodPicker mp = new MethodPicker();

int x = 5;

mp.pickMe(x);

}

}

... the code "mp.pickMe(x)" would choose method A (exact primitive type) first, but if that wasn't there, it would then choose method B (a larger primitive type).  Likewise if both A and B weren't there, it would then choose method C (the exact wrapper type).  Method D (varargs of the exact primitive type) would be the next option available, followed by method E (varargs of a larger primitive type).

If method C was written as:

public void pickMe(Long x) {

}

... it would not be called since it's a Long and the primitive was an int (it will only match wrappers that are the exact type of the primitive).

**What about an array?**

Although varargs and arrays are accessed similarly, their creation from method arguments is different.  Specifically, an array parameter is not automatically created from a one or more values... therefore, this method:

public void pickMe(int[] x) {

}

...would never be invoked from a single value  mp.pickMe(5);

or from multiple, such as values mp.pickMe(5,2,6,10); either.  You would need to pass in an actual array for a method with an array parameter to be invoked.

**One final rule**

An array and varargs of the same type are treated as if they are same parameter.  In other words, having the following two methods appear together in the same class is illegal and would not compile:

public void pickMe(int[] myArray) {

}

public void pickMe(int... myVarargs) {

}

**Chapter 17 – “Diamond Operator” Lecture**

**Building a list from Arrays.asList and Anonymous Arrays**

The Arrays class has a helper method to turn a varargs collection into a List:  Arrays.asList(T... elements);

So, an easy way to create a list of MyDate objects would be:

List<MyDate> dates = Arrays.asList(new MyDate(2,1,2020), new MyDate(3,10,1980));

Since a method that declares a vararg parameter may also accept an array argument, the following is also legal:

MyDate[] datesArray = { new MyDate(2,1,2020), new MyDate(3,10,1980) };

List<MyDate> dates = Arrays.asList(datesArray));

As we learned in the Arrays chapter, it's illegal to use the inline array initialization syntax if it's not part of the same statement as the variable declaration.  So the following would be illegal:

// ILLEGAL

List<MyDate> dates = Arrays.asList(

{ new MyDate(2,1,2020), new MyDate(3,10,1980) } );

However, there is a similar syntax, known as an anonymous array, which does work... (note the new MyDate[] in front of the initialization syntax)

List<MyDate> dates = Arrays.asList(

new MyDate[]{ new MyDate(2,1,2020), new MyDate(3,10,1980) } ); // LEGAL

Also as a note for the exam, remember that asList is a method in the Arrays class.  It's not found in the Collections class, nor is it found as a method in the list itself... so the following methods are illegal:

// ILLEGAL

List<MyDate> dates = Collections.asList(

new MyDate(2,1,2020), new MyDate(3,10,1980));

// LEGAL

List<MyDate> dates2 = new ArrayList<>();

// ILLEGAL

dates2 = dates2.asList(

new MyDate(2,1,2020), new MyDate(3,10,1980));

**Chapter 17 – “Comparator” Lecture**

**Type Parameters and Comparable/Comparator**

Just as we can add type parameters to our collections (thus eliminating the need to cast reference variables), we can also add them to the Comparable<T> and Comparator<T> interfaces.  Whatever type you replace T with is used in the compareTo(T t) and compare(T t1, T t2) methods respectively.

Here's an example of using Comparable with a type parameter (note that the type is Comparable<MyDate> and the compareTo method defines a MyDate parameter.

public class MyDate implements Comparable<MyDate> {

private int month;

private int day;

private int year;

public int compareTo(MyDate date) {

int result = 0;

if( year != date.year ) {

result = year - date.year;

} else if( month != date.month ) {

result = month - date.month;

} else if( day != date.day ) {

result = day - date.day;

}

return result;

}

}

... and here's an example of Comparator.  Note the type parameter listed with Comparator<MyDate> and that the compare method includes MyDate parameters.

import java.util.Comparator;

public class MyDateComparator implements Comparator<MyDate> {

public int compare(MyDate date1, MyDate date2) {

int result = 0;

if( date1.getYear() != date2.getYear() ) {

result = date1.getYear() - date2.getYear();

} else if( date1.getMonth() != date2.getMonth() ) {

result = date1.getMonth() - date2.getMonth();

} else if( date1.getDay() != date2.getDay() ) {

result = date1.getDay() - date2.getDay();

}

return result;

}

}