CS415

INTRODUCTION TO COMPUTER SCIENCE

FALL 2017

11 CONDITIONAL STATEMENTS

CHAPTER 6 AND 10

LAST TIME

- Numbers and operators:
  - Integers
  - Floating point
- Constants
- Class variables and methods
- The Math class

PREVIEW

- Conditions: boolean expressions
  - Relational operators
  - Boolean operators
- Conditionals Statements
  - if/then
  - if/then/else
  - switch

BOOLEAN TYPE

- Java provides a primitive type Boolean to model the values *true* and *false*.
  - Boolean values: true, false
  - Boolean variables:

```
boolean  batteryLow;  // declare a variable

batteryLow = false;      // assign a value to it
```

RELATIONAL OPERATORS

- Java provides binary relational operators
- The operators take numbers for arguments and produce boolean values.

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

PRECEDENCE SO FAR

Operator	
++, --, - (unary), (type)	unary
*, /, %	multiplicative
+, -	additive
<, >, <=, >=	relational
==, !=	equality
=, +=, -=, *=, /=, %=	assignment

CONDITIONS

- We can now form expressions with values *true* or *false* (Boolean expressions)
- Boolean expressions are also called conditions

```
roverBatteryCharge > 82.5

Math.abs( y – x ) < 0.001

count % 2 == 0

3 * 35 / 4.0  >= 28
```

- Be careful, it is a common error to confuse the assignment operator = and the equality operator ==

EQUALITY AND REFERENCES

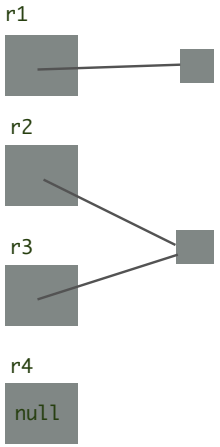
- The equality operators == and != can also be used with reference variables, although it is seldom what you want to do:
  - When used with a reference variable, the memory addresses referenced by the two variables are compared.

EQUALITY OPERATOR AND REFERENCE VARIABLES

```
Rectangle r1 = new Rectangle( 20, 20 );
Rectangle r2 = new Rectangle( 20, 20 );
Rectangle r3 = r2;
Rectangle r4;   // initialized to null

...

  r1 == r2    // false
  r2 == r3    // true
  r4 == null  // true
  r3 != null  // true
```



LOGICAL OPERATORS

- Logical operators take boolean expressions as arguments and produce boolean values.
- There are two binary logical operators
  && (and) and || (or)
- And one unary operator ! (not)

LOGICAL NOT

- Logical operators can be defined using *truth tables*.
- A truth table shows the value of the expression for all possible values of the arguments

condition	! condition
true	false
false	true

LOGICAL AND

- Since && (and) has two arguments there are four possible combinations of arguments

left	right	left && right
false	false	false
false	true	false
true	false	false
true	true	true

LOGICAL OR

- Note that || (or) is the "inclusive" or

left	right	left || right
false	false	false
false	true	true
true	false	true
true	true	true

PRECEDENCE SO FAR

Operator	
-, (type), !	unary
*, /, %	multiplicative
+, -	additive
<. >. <=, >=	relational
==, !=	equality
&&	logical and
||	logical or
=, <op>=	assignment

! AND THE RELATIONAL OPERATORS

```
!( a < b )   has the same value as   a >= b
!( a <= b )  has the same value as   a > b
!( a > b )   has the same value as   a <= b
!( a >= b )  has the same value as   a < b
!( a == b )  has the same value as   a != b
```


! AND BOOLEAN OPERATORS

- DeMorgan's rules tell us how to negate a boolean expression containing AND or OR

```
!( a && b )  has the same value as  !a || !b
!( a || b )  has the same value as  !a && !b
```

CONDITIONAL STATEMENTS

- A conditional statement lets us choose which statements will be executed based on a condition.
- Conditional statements give us the ability to make decisions.
- In Java the conditional statements are the *if-then* statement, the *if-then-else* statement, and the *switch* statement

THE IF-THEN STATEMENT

- Syntax of the if-then statement:

```
if ( <boolean expression> )
   < statement >
```

- Semantics of the if-then statement:
  - The <boolean expression> is evaluated
    - If the expression is *true,* execute <statement>
    - If the expression is *false*, don't execute <statement>

THE IF-THEN STATEMENT

- Example:

```
if ( shape != null )
   shape.hide( );
```

- Semantics:
  - If shape references an object (is not a *null* reference) then send it the hide message.
  - Otherwise do nothing.
- We are modeling the decision either to do something or to skip it.

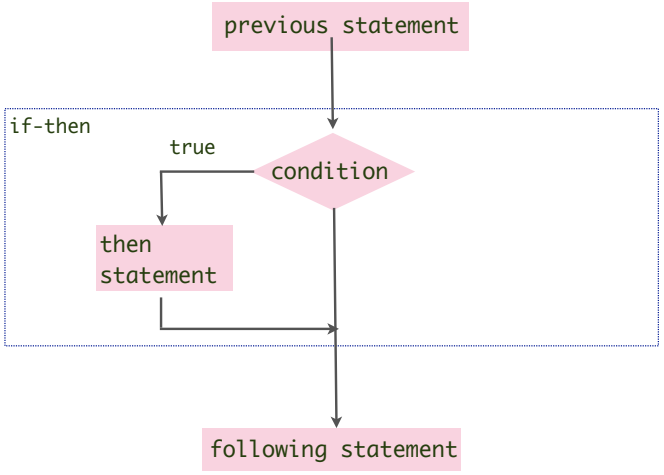
COMPOUND STATEMENTS

- A compound statement is a sequence of statements enclosed in brackets, { ... }
- You can use a compound statement anywhere that a simple statement is required.

```
if ( shape != null )
{
   shape.hide( );
   shape = null;
}
```

- If *shape* is not a null reference then hide it and make it a null reference.

FLOW CHART OF AN IF-THEN STATEMENT



THE IF-THEN-ELSE STATEMENT

- Syntax of the *if-then-else* statement:

```
if (<boolean expression>)
    <thenStatement>
else
    <elseStatement>
```

- Semantics of the *if-then-else* statement:
  - The boolean expression is evaluated
    - If the expression is *true*, execute <thenStatement>
    - If the expression is *false*, execute <elseStatement>

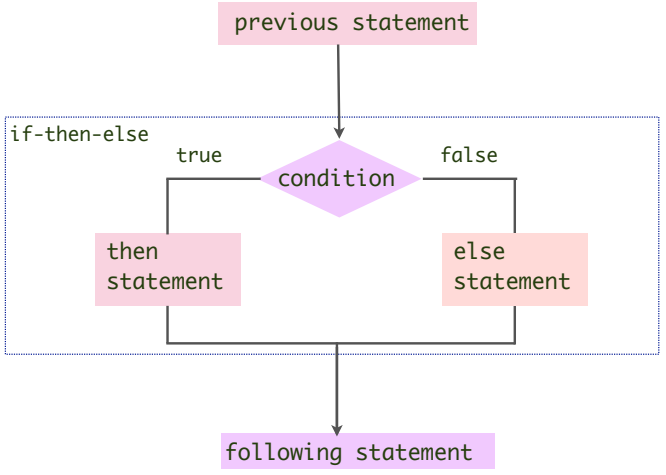
THE IF-THEN-ELSE STATEMENT

- Example:

```
if ( shape != null )
{
    shape.hide( );
    shape = null;
}
else
{
    System.out.println( “Shape is already null” );
}
```

- We are modeling a decision between two alternatives.

FLOW CHART OF AN IF-THEN-ELSE STATEMENT



CONDITIONAL STATEMENT STYLE CONVENTIONS

- The *if*, *else* and their opening and closing braces line up in the current indentation level column.
- The <then-statement> and <else-statement> are indented one level.

```
if ( shape != null )
{
   shape.hide( );
   shape = null;
}
else
{
   System.out.println( “Shape is already null” );
}
```


NESTED CONDITIONAL STATEMENTS

- A conditional statement is a <statement>
- Therefore, it can appear as a <then-statement> or an <else-statement> in a conditional statement

```
if ( shape == null )
    System.out.println( “Shape is null” );

else
    if ( shape.getColor() == Color.BLUE )
        shape.setColor( Color.RED );

    else
        shape.setColor( Color.BLUE );
```


NESTED CONDITIONAL STYLE

- When the conditional is nested as an <else-statement>:
  - Place the nested if on the line with the nesting else.
  - Indent the statements at the same level

```
if ( shape == null )
    System.out.println( “Shape is null” );

else if ( shape.getColor() == Color.BLUE )
    shape.setColor( Color.RED );

else
    shape.setColor( Color.BLUE );
```


NESTED CONDITIONAL STYLE (MULTIPLE NESTING)

```
if ( trafficLight == null )
    System.out.println( “Traffic light is null” );

else if ( trafficLight.getColor() == Color.RED )
    trafficLight.setColor( Color.GREEN );

else  if ( trafficLight.getColor() == Color.GREEN )
    trafficLight.setColor( Color.YELLOW );

else if ( trafficLight.getColor() == Color.YELLOW )
    trafficLight.setColor( Color.RED );

else
    System.out.println( “Traffic light color error” )
```

USING PREDICATE METHODS

- A function method that returns a boolean value is called a predicate method
- For example, a wheels Shape has a predicate method *public boolean contains(Point p)*

```
Point p = new Point( 50, 50 );
Ellipse circle = new Ellipse( );

if ( circle.contains( p ) )
    circle.setColor( Color.WHITE );
```

- The following is unnecessary and bad style

```
if ( circle.contains( p ) == true )
```


WRITING PREDICATE METHODS

- Write a predicate method to tell if two shapes are located at the same point

```
public class Box extends Rectangle
{
   ...
   public boolean isCoincident( Rectangle r )
   {
      return this.getXLocation() == r.getXLocation() &&
             this.getYLocation() == r.getYLocation();
   }
   ...
}
```

- The following is unnecessary and bad style

```
if (      this.getXLocation() == r.getXLocation()
     &&  this.getYLocation() == r.getYLocation() )
   return true;
else
   return false;
```


BOOLEAN VARIABLES

- Just like numerical values, it is sometimes convenient to save a boolean value in a variable:

```
boolean  patientHealthy;  // default value false
...
patientHealthy = !patient.hasInsurance() && pulse > 0;
...

if ( patientHealthy )
    patient.discharge();
```


BOOLEAN VARIABLES

```
if ( patientHealthy )
    patient.discharge();
```

- The following is unnecessary and bad style

```
if ( patientHealthy == true )
    patient.discharge();
```

- It can also lead to a common error; what's wrong with the following code?

```
alert = NORAD.attackConfirmed( );
if ( alert = true )
    missles.launchCounterAttack( );
```

RANDOM NUMBERS

- Sometimes in a program, a game for example, we might want to have some "random" behavior.
  - flip a coin, roll the dice, deal a card.
- Java provides methods to generate "pseudo-random" number sequences.
- A pseudo-random sequence is a sequence that "appears random" even though it is generated by deterministic process.

RANDOM NUMBER SEEDS

- A pseudo-random generator begins with an initial value called a seed.
- The sequence of numbers generated with a given seed will always be the same.
- This is useful for testing.
- To get different sequences generated you start with different seeds.
- Often the time on the system clock is used as a seed (after testing is done).

JAVA.UTIL.RANDOM

- First create a (pseudo)random generator:

```
import java.util.Random;

Random gen1 = new Random( 12345 ); // seed for testing
Random gen2 = new Random( ); // seed based on system clock
```

- You can now generate (pseudo)random values

```
double d = gen1.nextDouble( );      //    0.0 <= x   < 1.0

int i = gen1.nextInt( 6 );          //    0   <= i   < 6

int die = gen1.nextInt( 6 ) + 1;    //    1   <= die <= 6

float f = gen1.nextFloat( ) * 100;  //    0.0 <= f   < 100.0

boolean win = gen1.nextBoolean( );  //   win == true or win == false
```


THE SWITCH STATEMENT

- When a multi-way selection depends on the value of an integer (or character) variable, a *switch* selection statement is sometimes convenient.
- The following is the beginning of a *switch* statement that will make a selection based on the value of the variable *test*.

```
int test;
...
switch ( test )
{

}
```

THE SWITCH STATEMENT

- Selection is implemented with *case* labels that determine the entry point of the *switch* based on the value of the variable.
- The *default* case is the entry point if no other cases match.

```
int test;
...
switch ( test )
{
    case 1, 2: statementA;   // start here if test == 1 or test == 2
               statementB;
    case 3:    statementC;   // start here if test == 3

    default:   statementX;   // start here if no other cases match
}
```


THE SWITCH STATEMENT

- Once the *switch* is entered, execution continues to the end of the *switch* unless a *break* statement is encountered.
- Normally, each *case* will end with a *break*.

```
int test;
test = nextInt( 6 ) + 1; //  1 ... 6
switch ( test )
{
    case 1, 2: statementA;   // execute A and B
               statementB;
               break;

    case 3:    statementC;   // execute C
               break;

    case 4:    statementD;   // execute D
               break;

    default:   statementX;   // execute X if test is not 1,2,3 or 4
}
```


THE SWITCH STATEMENT

- In a case without a *break*, execution will flow into the next *case*.
- In the rare situation where you want this behavior, make it clear to the reader with a comment.

```
switch ( test )
{
    case 1, 2: statementA;   // execute A and B
               statementB;
               break;

    //=============  "flow-through" case without break ============
    case 3:    statementC;   // no break, execute C and D

    case 4:    statementD;   // execute D
               break;

    default:   statementX;   // execute X if test is not 1,2,3 or 4
}
```


JAVA KEY WORDS WE'VE SEEN

```
abstract        implements      try
continue        protected       char
for             throw           final
new             byte            interface
switch          else            static
assert          import          void
default         public          class
goto            throws          finally
package         case            long
synchronized    enum            strictfp
boolean         instanceof      volatile
do              return          const
if              transient       float
private         catch           native
this            extends         super
break           int             while
double          short
```

REVIEW

- Conditions: boolean expressions
  - Relational operators
  - Boolean operators
- Conditionals: Modeling Decisions
  - if/then
  - if/then/else
  - switch

NEXT TIME

- Chapter 11 Loops
  - Modeling repetition
  - Definite/indefinite loops
  - while, for, do-while loops
  - sentinel, best value, accumulator recipes
  - hand simulation of loops