| Type | Values or largest positive* | Size (bits) |
|---|---|---|
| boolean | true or false | 1 |
| char | 65,536 possibilities | 16 |
| byte | 127 | 8 |
| short | 32,767 | 16 |
| int | 2,147,483,647 | 32 |
| long | 9,223,372,036,854,775,807 | 64 |
| float | $\pm 10^{38}$, 7 decimal digits | 32 |
| double | $\pm 10^{308}$, 15 decimal digits | 64 |
| *largest negative is one higher than largest positive | | |

**Casting**

- Numbers in expressions will automatically be cast up to a type with greater precision; if either is float/double, the other will be cast to float/double
- To cast down to a type with less precision, precede the number/result by "(type)"

**Common Syntax**

public static void main(String args[ ]){ }
*accessSpecifier* static final *typeName variableName = value*;
in method: final *typeName variableName = value*;
*accessSpecifier* enum *TypeName* { *value1*, *value2*, ... }
using enum: *ClassName.TypeName.value*

| Operators | Highest Precedence |
|---|---|
| postfix | *expr*++, *expr*–– |
| unary | ++*expr*, ––*expr*, +*expr*, –*expr*, ~, ! |
| multiplicative | *, /, % |
| additive | +, – |
| shift | <<, >>, >>> |
| relational | <, <=, >, >=, instanceof |
| equality | ==, != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | \| |
| logical AND | && |
| logical OR | \|\| |
| conditional | ? : |
| assignment | =, +=, –=, *=, /=, %=, &=, ^=, \|=, <<=, >>=, >>>= |
| | **Lowest Precedence** |

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html

- && short circuits on false, || short circuits on true
- In division of two ints, fractional part is truncated.
- int % int returns remainder of integer division

| 1D array | 2D array |
|---|---|
| int[ ] arr = new int[3]; //int array<br>int lth = arr.length; //length of array<br>arr[0] = 1; // index = 0,..,lth-1, arr[index]<br>arr[1] = 3;<br>arr[2] = 5;<br>// equivalent to: arr = {1,3,5};<br>//reference type array<br>Object[ ] ob = new Object[4]; | // when every row has the same number of columns<br>int[ ][ ] arr2 = new int[# of rows][# of cols];<br>int r = arr2.length; // # rows of array<br>int c = arr2[2].length; // # cols in specific, say, 3rd row<br>// ragged array<br>int[ ][ ] arr3 = new int[# of rows][ ]; // leave second blank<br>arr3[0] = new int[# of cols in 1st row];<br>arr3[1] = new int[# of cols in 2nd row]; |

**Loops:** conditions are boolean expression

| for | while | do while |
|---|---|---|
| (run from starting to ending value with constant indrement/decrement)<br>int[ ] arr = {1,3,5}<br>// for(initialize; condition; update)<br>// statement<br>for( int i =0; i < arr.length; i++){<br>//scope of int, i<br>System.out.println(arr[i]); } | (statements executed condition is false.)<br>// while(condition)<br>// statement<br>// while version of a for loop:<br>int i = 0;<br>while(i < arr.length){<br>System.out.println(arr[i]);<br>i++; //update} | (always executed once)<br>//do{<br>// statements<br>//}while(condition);<br>do{<br>System.out.println(…);<br>}while(condition); |

**Branching:** different actions depending on different conditions (can use boolean expressions)

| | | |
|---|---|---|
| if(condition1){<br>statement1;<br>}else if(condition2){<br>statement2;<br>}…<br>else{<br>statement;<br>}<br>e.g<br>if(richter >= 6.0)<br>r= "heavily damaged";<br>else if(richter >= 3.5)<br>r= "lightly damaged";<br>else if(richter >= 0)<br>r= "ok"; | switch-case: if/else if/ else when comparing single int or char values<br>int d;<br>string s = " ";<br>switch(d){<br>case 1: s= "one"; break; // d==1<br>case 2: s= "two"; break; //d==2<br>default: s= "na"; //otherwise }<br>/* "break" each case if alternatives are exclusive, otherwise excuted cumulatively. */<br>/* "continue" jump to the end of the loop body*/ | making comparisons<br>1) floating pt numbers: check that they are close enough, instead of equal.<br>final double EPS = 1E-15;<br>if(Math.abs(x-y) <= EPS){…};<br>2) strings<br>//if both refer to the identical object<br>if(string1==string2)<br>if(string1.equals(string2)) //if both equal<br>//dictionary order, compareTo return 0 if<br>//equal, negative if string1 comes before<br>//string2, positive otherwise:<br>if(string1.compareTo(string2) <0)<br>3) object : "==" tests whether references refer to the same object. Use equal method to compare the contents |

**1. Throwing Exceptions:**

General Format:

```
if (condition)
throw new exception_type(param0, param1, …);
```

Throws Clause:

```
public void method() throws exception_type1, exception_type2, ... {
//These exceptions are caught in the catch statement later on.
}
```

**2. Catching Exceptions:**

- Catch the most specific exceptions first, then broader exceptions.
- A while loop at the beginning a try-catch block is a good way to continuously prompt the user for input in case they enter the wrong information.

**3. Making your own Exceptions:**

example:

```
if (amount > balance){
throw new InsufficientFundsException(
"withdrawal of " + amount + " exceeds balance of " + balance);
```

Defining InsufficientFundsException class:

```
public class InsufficientFundsException extends RuntimeException{
   public InsufficientFundsException(){
   }
   public InsufficientFundsException(String message){
      super(message);
   }
}
```

**General Format of try-catch block**

```
try{
   Main_block_of_statements
}
catch (exception_type1 variable1){
   Block_of_statements
}
catch (exception_type2 variable2){
   block_of_statements2
}
//more catch blocks if necessary
finally{
/* Finally block is optional.  Statements in this block are always executed  When might you need a finally block?  When you close a file reader.*/
}
```

**4. Checked vs. Unchecked Exceptions:**

Checked exceptions must be handled by your program:

Exception←ClassNotFoundException, IOException←EOException, FileNotFoundException

Unchecked exceptions:

RuntimeException←NullPointerException, IndexOutOfBoundsException←ArrayIndexOutOfBoundsException

**Inheritance**

- Allows the design of general class (super class) that can be specialized in more specific classes (sub class).
- The sub class *extends* the super class: public class *SubclassName* extends *SuperClassName*
- Only single inheritance for sub classes is allowed.
- Sub classes can override methods from the super class.
- If a super constructor is not called, the super constructor without parameter (one must exist) is called automatically.
- "this" refers to the current instance of the object; "super" refers to the parent class of the current subclass object.

**Polymorphism**

- The ability of an object variable to take different forms.
- Also found in overloading of method and constructor names- return types must be the same but number of parameters and/or parameter types must be different.
- If a method called on an reference variable of superclass X that points to an object of subclass Y (that extends X), Y's method will be called instead of X's.  Y's method is then said to *override* X's.

**Class Casting**

- A superclass reference variable my point to a subclass object.  Only methods in superclass X can be called on a reference variable of type X even if the reference points to a subclass that may have additional methods.
- If the object pointed to by a superclass reference variable is actually a subclass, the superclass reference variable can be cast to the subclass: *SubClassName variableName* = (*SubClassName*) *superClassVariableName*.  Use "instanceof" to check that the object is the correct subclass
- Cannot cast between subclasses of the same superclass (siblings).

**Interfaces**

- Outline for a class.
- Contains method signatures but does not tell how the methods are implemented.
- General form: public interface *InterfaceName*
- A class that implements an interface is required to implement all the methods listed in the interface.
- A class can implement more than one interface.
- General form: public class *ClassName* implements *InterfaceName*