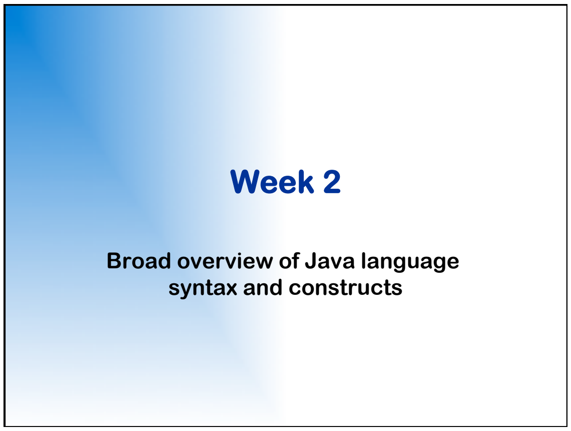
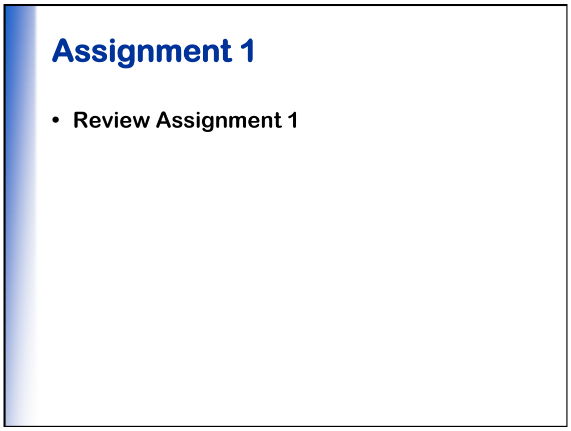


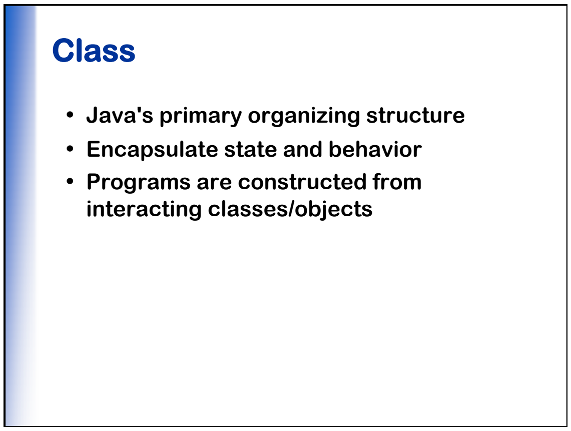
Week 2

Broad overview of Java language syntax and constructs



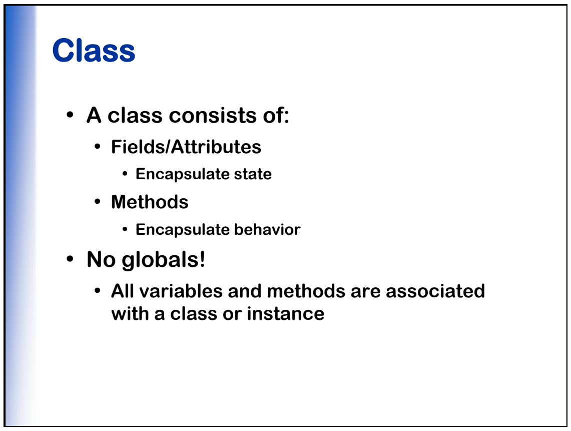
Assignment 1

- Review Assignment 1



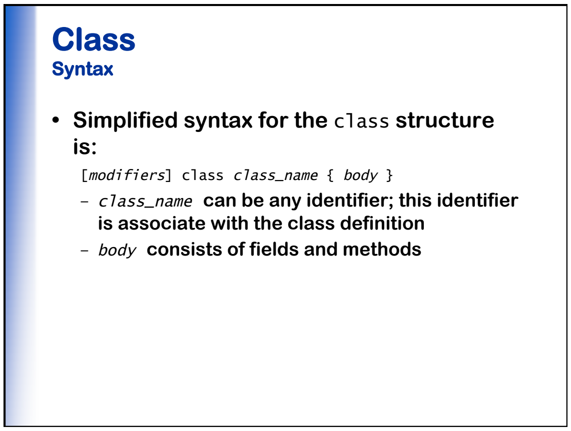
Class

- Java's primary organizing structure
- Encapsulate state and behavior
- Programs are constructed from interacting classes/objects



Class

- A class consists of:
  - Fields/Attributes
    - Encapsulate state
  - Methods
    - Encapsulate behavior
- No globals!
  - All variables and methods are associated with a class or instance

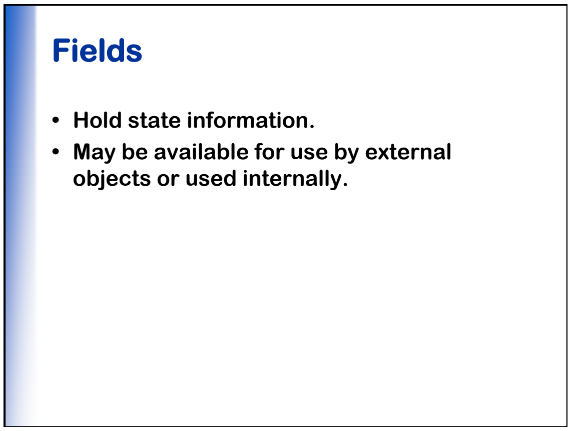


Class
Syntax

- Simplified syntax for the `class` structure is:

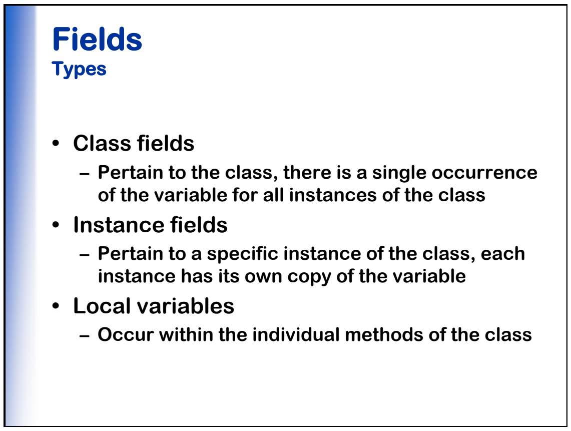
```
[modifiers] class class_name { body }
```

  - *`class_name`* can be any identifier; this identifier is associate with the class definition
  - *`body`* consists of fields and methods



Fields

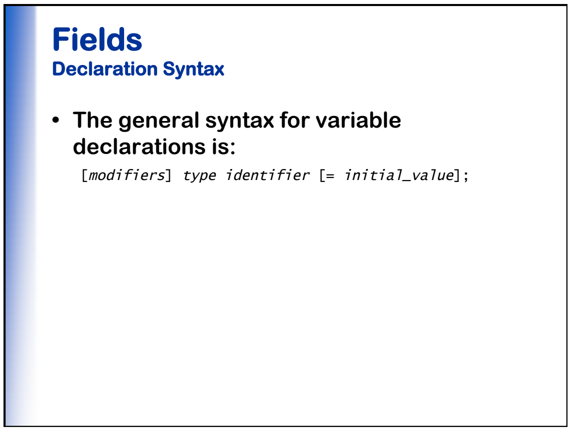
- Hold state information.
- May be available for use by external objects or used internally.



Fields

Types

- Class fields
  - Pertain to the class, there is a single occurrence of the variable for all instances of the class
- Instance fields
  - Pertain to a specific instance of the class, each instance has its own copy of the variable
- Local variables
  - Occur within the individual methods of the class

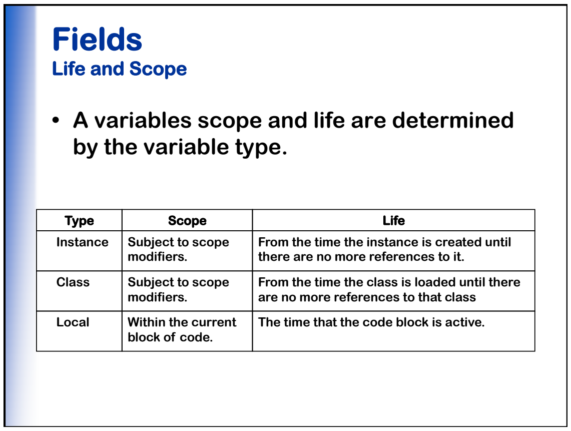


Fields

Declaration Syntax

- The general syntax for variable declarations is:

```
[modifiers] type identifier [= initial_value];
```

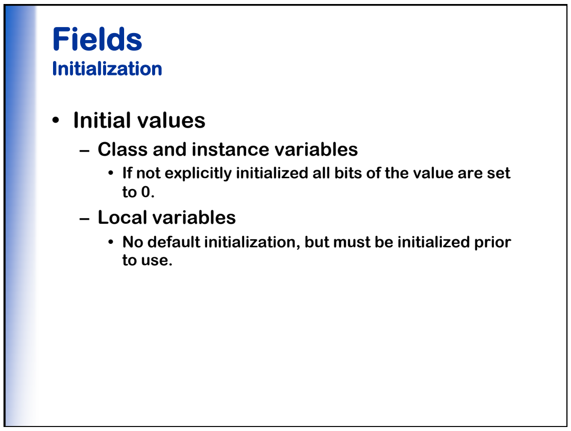


Fields

Life and Scope

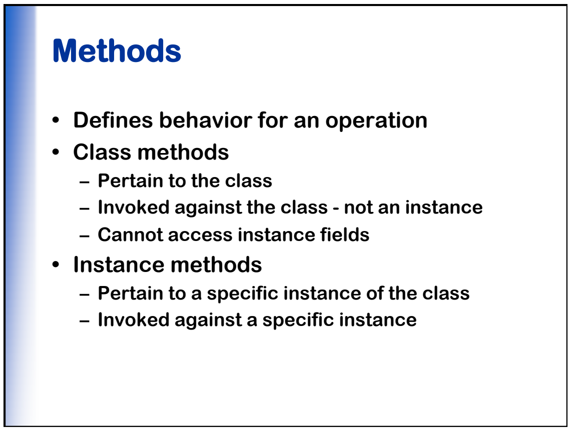
- A variables scope and life are determined by the variable type.

Type	Scope	Life
Instance	Subject to scope modifiers.	From the time the instance is created until there are no more references to it.
Class	Subject to scope modifiers.	From the time the class is loaded until there are no more references to that class
Local	Within the current block of code.	The time that the code block is active.



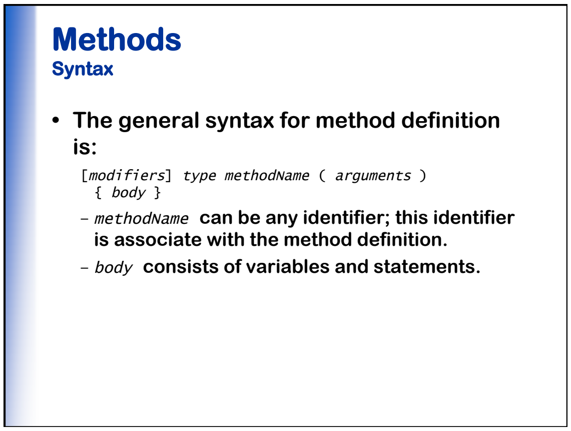
Fields
Initialization

- Initial values
  - Class and instance variables
    - If not explicitly initialized all bits of the value are set to 0.
  - Local variables
    - No default initialization, but must be initialized prior to use.



Methods

- Defines behavior for an operation
- Class methods
  - Pertain to the class
  - Invoked against the class - not an instance
  - Cannot access instance fields
- Instance methods
  - Pertain to a specific instance of the class
  - Invoked against a specific instance

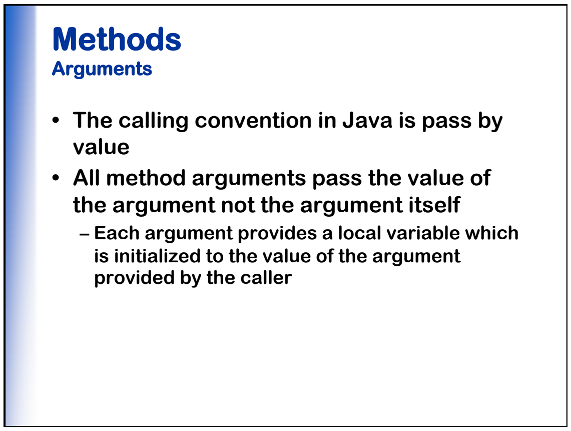


Methods
Syntax

- The general syntax for method definition is:

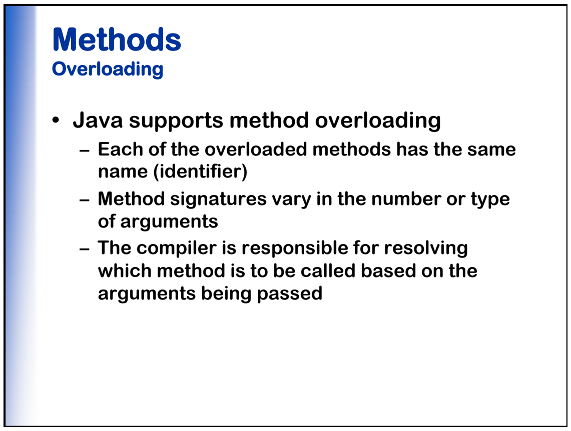
```
[modifiers] type methodName ( arguments )
  { body }
```

  - *methodName* can be any identifier; this identifier is associate with the method definition.
  - *body* consists of variables and statements.



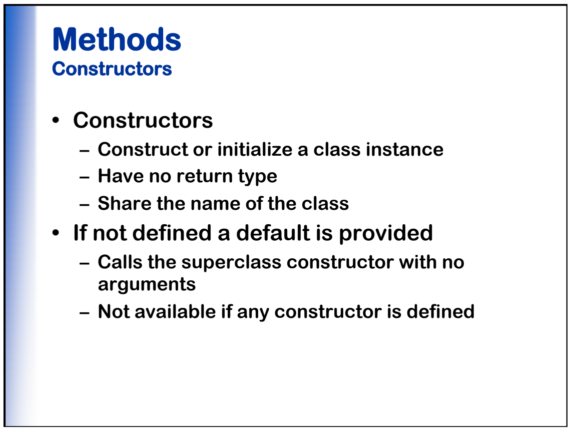
Methods
Arguments

- The calling convention in Java is pass by value
- All method arguments pass the value of the argument not the argument itself
  - Each argument provides a local variable which is initialized to the value of the argument provided by the caller



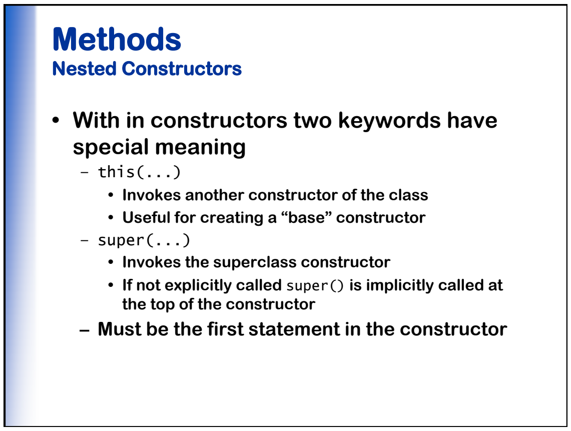
Methods
Overloading

- Java supports method overloading
  - Each of the overloaded methods has the same name (identifier)
  - Method signatures vary in the number or type of arguments
  - The compiler is responsible for resolving which method is to be called based on the arguments being passed



Methods
Constructors

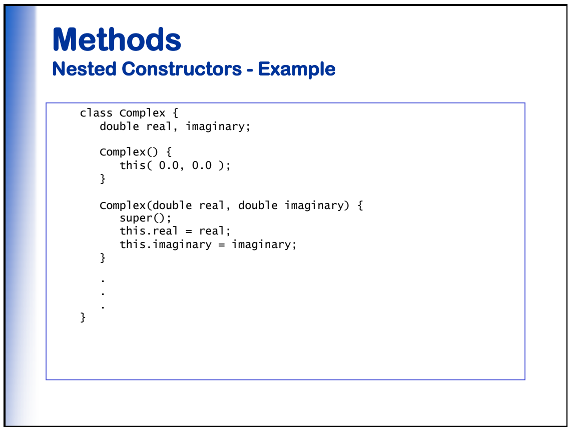
- Constructors
  - Construct or initialize a class instance
  - Have no return type
  - Share the name of the class
- If not defined a default is provided
  - Calls the superclass constructor with no arguments
  - Not available if any constructor is defined



Methods

Nested Constructors

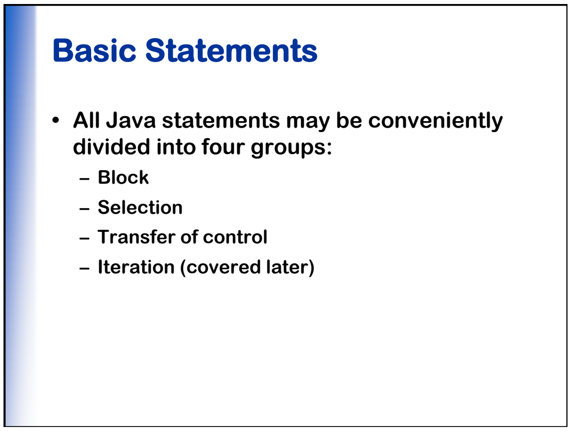
- With in constructors two keywords have special meaning
  - `this(...)`
    - Invokes another constructor of the class
    - Useful for creating a “base” constructor
  - `super(...)`
    - Invokes the superclass constructor
    - If not explicitly called `super()` is implicitly called at the top of the constructor
  - Must be the first statement in the constructor



Methods

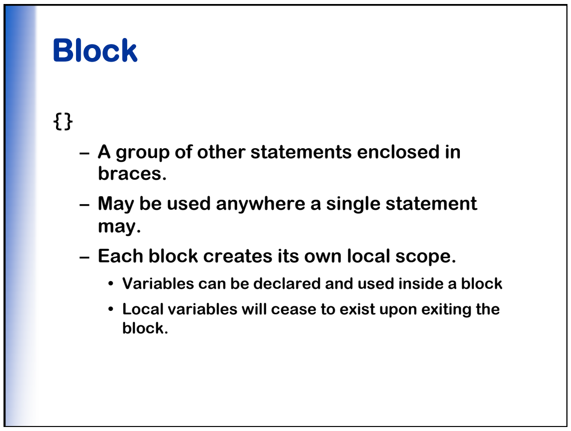
Nested Constructors - Example

```
class Complex {
   double real, imaginary;

   Complex() {
      this( 0.0, 0.0 );
   }

   Complex(double real, double imaginary) {
      super();
      this.real = real;
      this.imaginary = imaginary;
   }

   .
   .
   .
}
```



Basic Statements

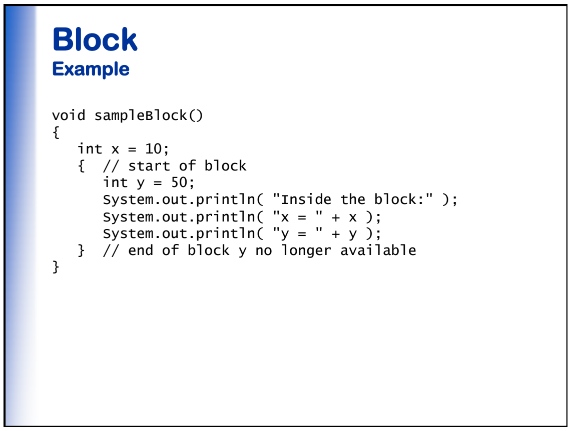
- All Java statements may be conveniently divided into four groups:
  - Block
  - Selection
  - Transfer of control
  - Iteration (covered later)



Block

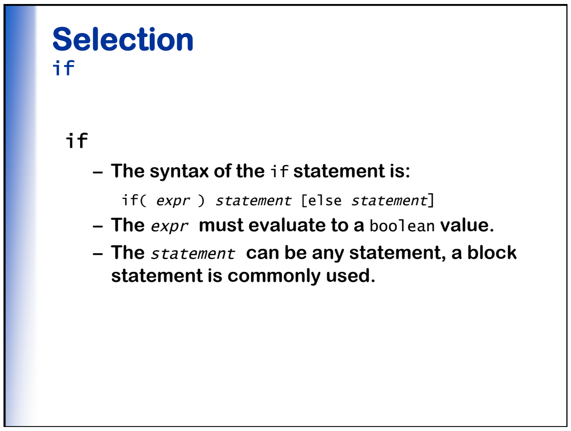
{}

- A group of other statements enclosed in braces.
- May be used anywhere a single statement may.
- Each block creates its own local scope.
  - Variables can be declared and used inside a block
  - Local variables will cease to exist upon exiting the block.



Block
Example

```
void sampleBlock()
{
   int x = 10;
   {  // start of block
      int y = 50;
      System.out.println( "Inside the block:" );
      System.out.println( "x = " + x );
      System.out.println( "y = " + y );
   }  // end of block y no longer available
}
```



Selection
`if`

`if`

- The syntax of the `if` statement is:

  `if( `*`expr`*` ) `*`statement`*` [else `*`statement`*`]`
- The *`expr`* must evaluate to a `boolean` value.
- The *`statement`* can be any statement, a block statement is commonly used.

Selection
`if` vs. Conditional Operator

- The conditional operator is very similar to the `if` statement
- The conditional operator yields an expression. The syntax of the conditional operator is as follows:

```
( testExpr ) ? trueExpr : falseExpr
```

Selection
Examples

- The following code fragments are equivalent.

```
if( x > y ) max = x; else max = y;
```

- or –

```
max = x > y ? x : y;
```

- Can't do this with `if`

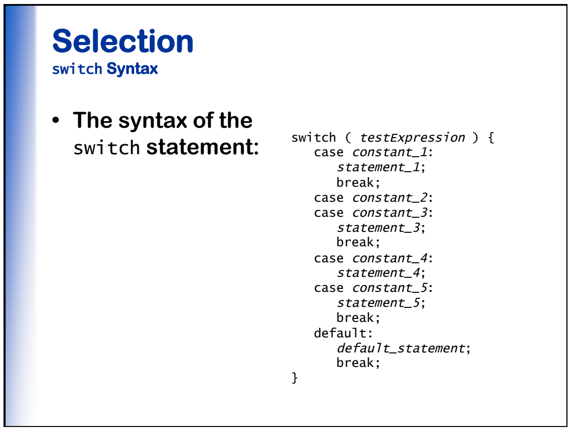
```
max = if( x > y ) x; else y;
```

  - If statements are not expressions

Selection
`switch`

`switch`

- Allows testing a single `int` (or `enum`) expression against a number of values, if a matching value is found the statement associated with that value is executed.
- The `default` statement which is executed in the event none of the values match the expression.
- The `break` statement is required to prevent fall-through.

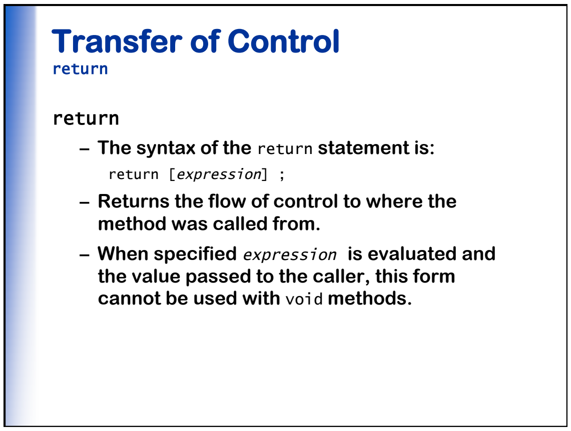


Selection

switch Syntax

- The syntax of the switch statement:

```
switch ( testExpression ) {
   case constant_1:
      statement_1;
      break;
   case constant_2:
   case constant_3:
      statement_3;
      break;
   case constant_4:
      statement_4;
   case constant_5:
      statement_5;
      break;
   default:
      default_statement;
      break;
}
```



Transfer of Control

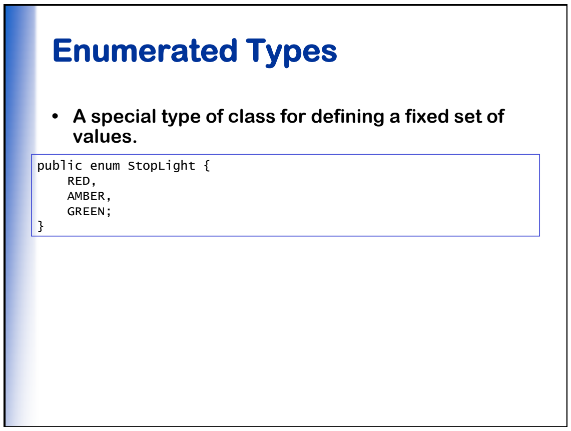
return

return

- The syntax of the return statement is:

  ```
  return [expression] ;
  ```

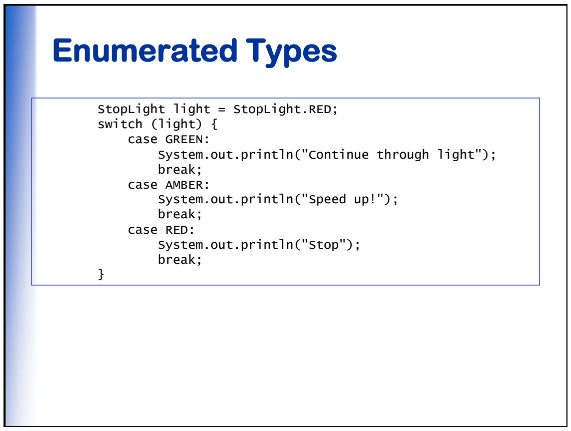
- Returns the flow of control to where the method was called from.
- When specified *expression* is evaluated and the value passed to the caller, this form cannot be used with void methods.



Enumerated Types

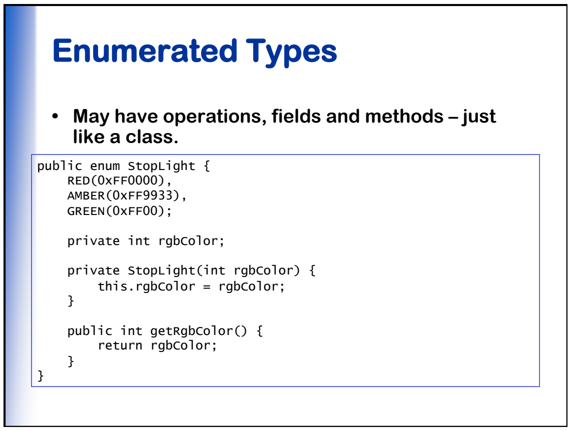
- A special type of class for defining a fixed set of values.

```
public enum StopLight {
    RED,
    AMBER,
    GREEN;
}
```



Enumerated Types

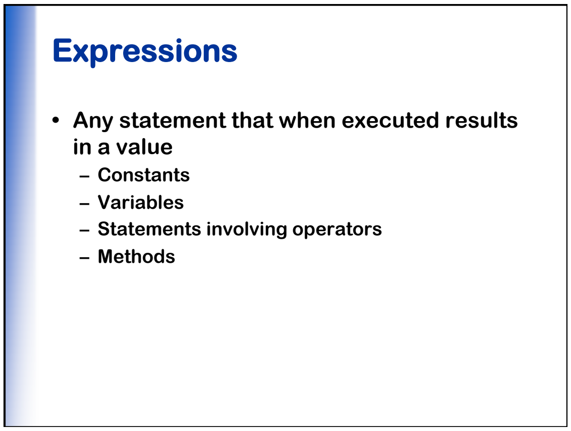
```
StopLight light = StopLight.RED;
switch (light) {
    case GREEN:
        System.out.println("Continue through light");
        break;
    case AMBER:
        System.out.println("Speed up!");
        break;
    case RED:
        System.out.println("Stop");
        break;
}
```



Enumerated Types

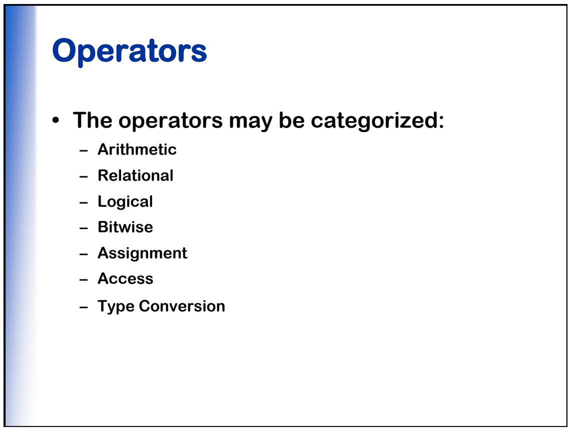
- May have operations, fields and methods – just like a class.

```
public enum StopLight {
    RED(0xFF0000),
    AMBER(0xFF9933),
    GREEN(0xFF00);

    private int rgbColor;

    private StopLight(int rgbColor) {
        this.rgbColor = rgbColor;
    }

    public int getRgbColor() {
        return rgbColor;
    }
}
```



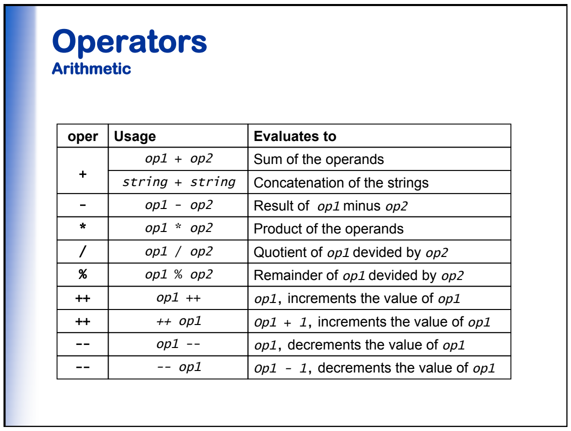
Expressions

- Any statement that when executed results in a value
  - Constants
  - Variables
  - Statements involving operators
  - Methods



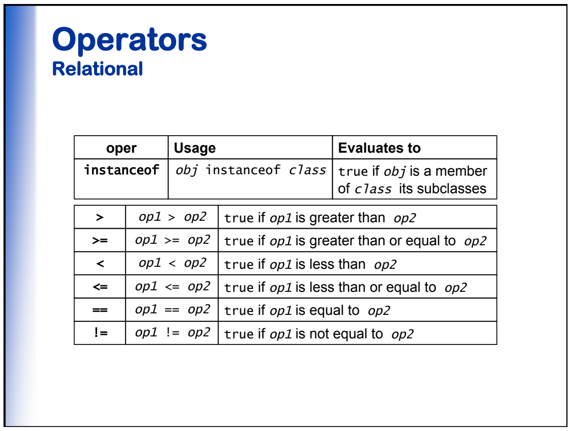
Operators

- The operators may be categorized:
  - Arithmetic
  - Relational
  - Logical
  - Bitwise
  - Assignment
  - Access
  - Type Conversion



Operators
Arithmetic

oper	Usage	Evaluates to
+	*op1* + *op2*	Sum of the operands
	string + *string*	Concatenation of the strings
-	*op1* - *op2*	Result of *op1* minus *op2*
*	*op1* * *op2*	Product of the operands
/	*op1* / *op2*	Quotient of *op1* devided by *op2*
%	*op1* % *op2*	Remainder of *op1* devided by *op2*
++	*op1* ++	*op1*, increments the value of *op1*
++	++ *op1*	*Op1* + *1*, increments the value of *op1*
--	*op1* --	*op1*, decrements the value of *op1*
--	-- *op1*	*Op1* - *1*, decrements the value of *op1*



Operators
Relational

oper	Usage	Evaluates to
instanceof	*obj* instanceof *class*	true if *obj* is a member of *class* its subclasses

>	*op1* > *op2*	true if *op1* is greater than *op2*
>=	*op1* >= *op2*	true if *op1* is greater than or equal to *op2*
<	*op1* < *op2*	true if *op1* is less than *op2*
<=	*op1* <= *op2*	true if *op1* is less than or equal to *op2*
==	*op1* == *op2*	true if *op1* is equal to *op2*
!=	*op1* != *op2*	true if *op1* is not equal to *op2*

Operators
Logical

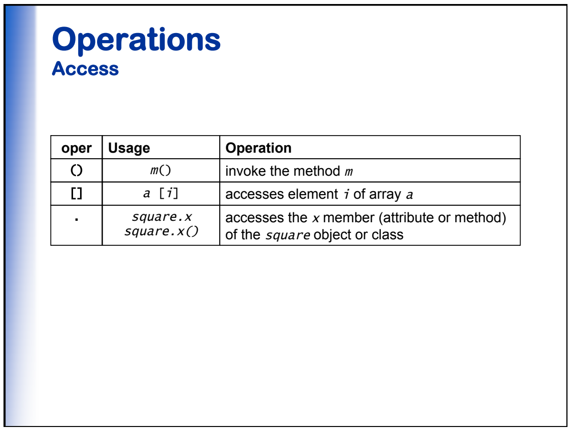
oper	Usage	Evaluates to
&&	*exp1* && *exp2*	true if *exp1* AND *exp2* are true
||	*exp1* || *exp2*	true if *exp1* OR *exp2* is true
!	! *boolExpr*	Negation of the expression *boolExpr*
?:	*boolExpr* ? *e1* : *e2*	*e1* if *boolExpr* is true otherwise *e2*

Operators
Bitwise

oper	Usage	Operation
>>	*op1* >> *op2*	shifts the bits of *op1*, *op2* places to the right, fills with the sign bit
<<	*op1* << *op2*	shifts the bits of *op1*, *op2* places to the left
>>>	*op1* >>> *op2*	shifts the bits of *op1*, *op2* places to the right, fills with 0
&	*op1* & *op2*	result has bits set in *op1* AND *op2* set
|	*op1* | *op2*	result has bits set in *op1* OR *op2* set
^	*op1* ^ *op2*	result has bits set in *op1* OR *op2*, but not both, set
~	~ *op1*	value of *op1* with each bit toggled

Operators
Assignment

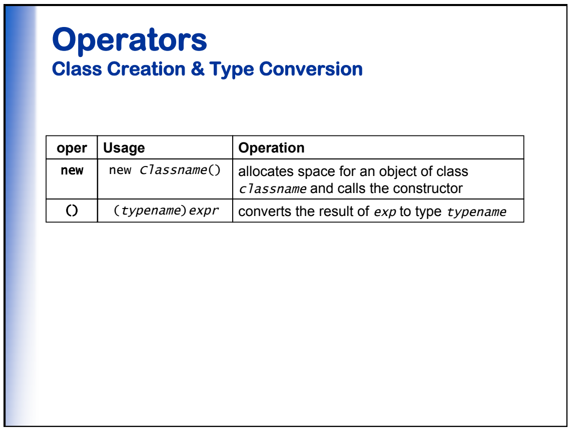
oper	Usage	Operation/Equivalent to
=	*var* = *exp*	assigns the value of the expression exp to var
+=	*op1* += *op2*	*op1* = *op1* + *op2*
-=	*op1* -= *op2*	*op1* = *op1* - *op2*
*=	*op1* *= *op2*	*op1* = *op1* * *op2*
/=	*op1* /= *op2*	*op1* = *op1* / *op2*
%=	*op1* %= *op2*	*op1* = *op1* % *op2*
&=	*op1* &= *op2*	*op1* = *op1* & *op2*
|=	*op1* |= *op2*	*op1* = *op1* | *op2*
^=	*op1* ^= *op2*	*op1* = *op1* ^ *op2*
<<=	*op1* <<= *op2*	*op1* = *op1* << *op2*
>>=	*op1* >>= *op2*	*op1* = *op1* >> *op2*
>>>=	*op1* >>>= *op2*	*op1* = *op1* >>> *op2*



Operations

Access

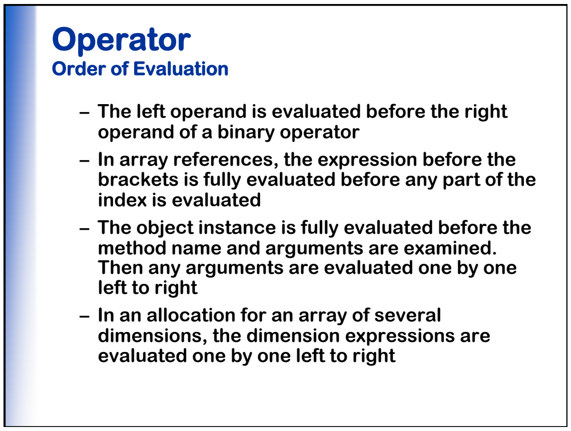
oper	Usage	Operation
()	*m*()	invoke the method *m*
[]	*a* [*i*]	accesses element *i* of array *a*
.	*square*.*x* *square*.*x*()	accesses the *x* member (attribute or method) of the *square* object or class



Operators

Class Creation & Type Conversion

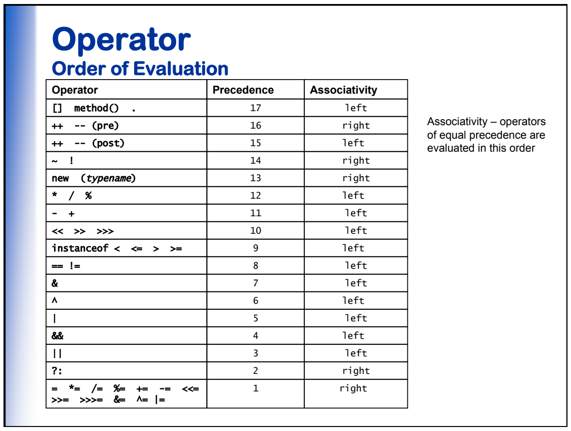
oper	Usage	Operation
new	new *Classname*()	allocates space for an object of class *classname* and calls the constructor
()	(*typename*)*expr*	converts the result of *exp* to type *typename*



Operator

Order of Evaluation

- The left operand is evaluated before the right operand of a binary operator
- In array references, the expression before the brackets is fully evaluated before any part of the index is evaluated
- The object instance is fully evaluated before the method name and arguments are examined. Then any arguments are evaluated one by one left to right
- In an allocation for an array of several dimensions, the dimension expressions are evaluated one by one left to right

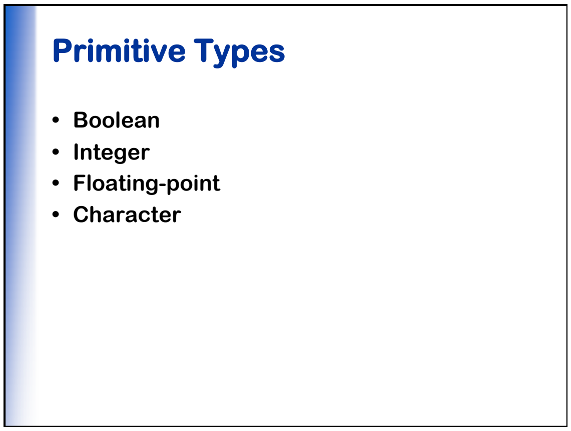


Operator

Order of Evaluation

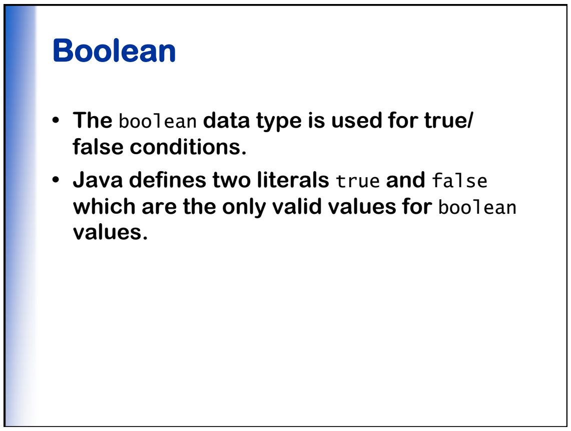
Operator	Precedence	Associativity
`[]  method()  .`	17	left
`++  -- (pre)`	16	right
`++  -- (post)`	15	left
`~  !`	14	right
new (*typename*)	13	right
`*  /  %`	12	left
`-  +`	11	left
`<<  >>  >>>`	10	left
`instanceof <  <=  >  >=`	9	left
`== !=`	8	left
`&`	7	left
`^`	6	left
`|`	5	left
`&&`	4	left
`||`	3	left
`?:`	2	right
`=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`	1	right

Associativity – operators of equal precedence are evaluated in this order



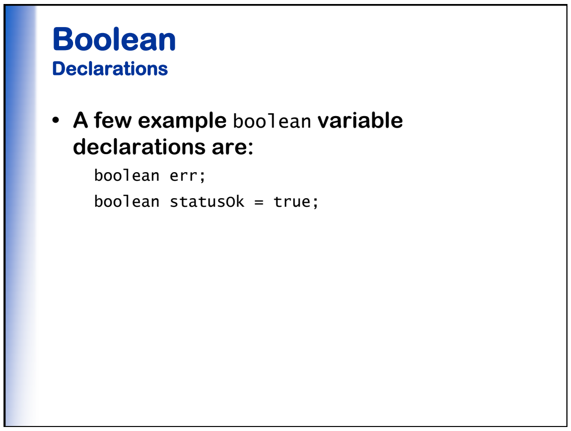
Primitive Types

- Boolean
- Integer
- Floating-point
- Character



Boolean

- The `boolean` data type is used for true/false conditions.
- Java defines two literals `true` and `false` which are the only valid values for `boolean` values.

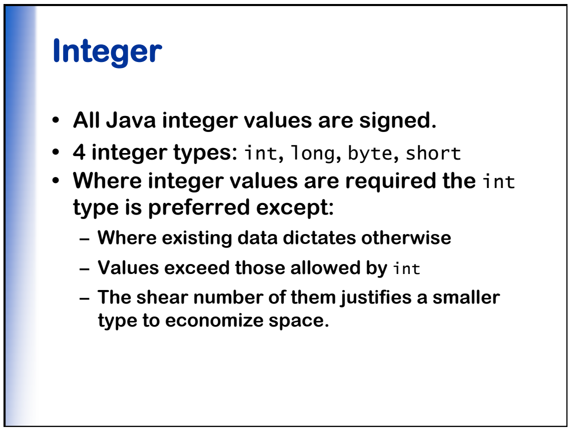


Boolean

Declarations

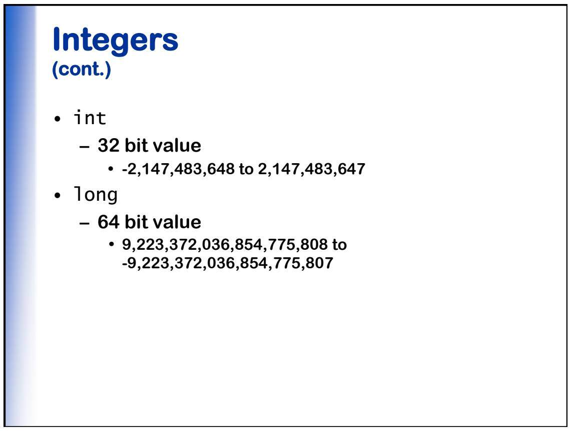
- A few example `boolean` variable declarations are:

```
boolean err;
boolean statusOk = true;
```



Integer

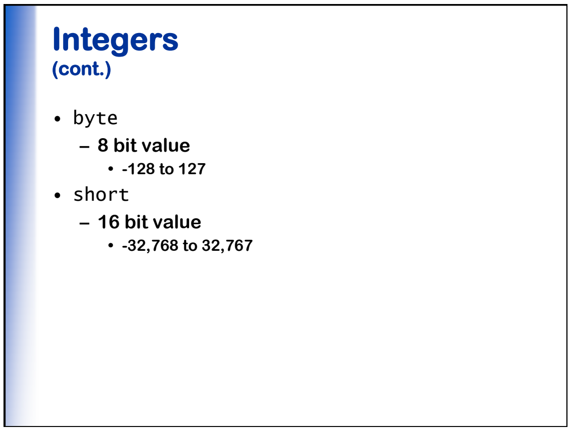
- All Java integer values are signed.
- 4 integer types: `int`, `long`, `byte`, `short`
- Where integer values are required the `int` type is preferred except:
  - Where existing data dictates otherwise
  - Values exceed those allowed by `int`
  - The shear number of them justifies a smaller type to economize space.



Integers

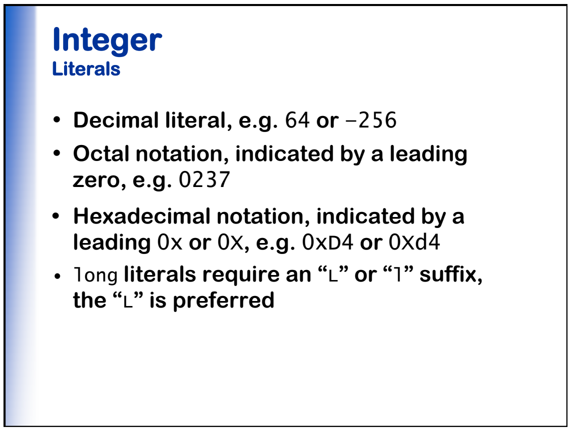
(cont.)

- `int`
  - 32 bit value
    - -2,147,483,648 to 2,147,483,647
- `long`
  - 64 bit value
    - 9,223,372,036,854,775,808 to -9,223,372,036,854,775,807



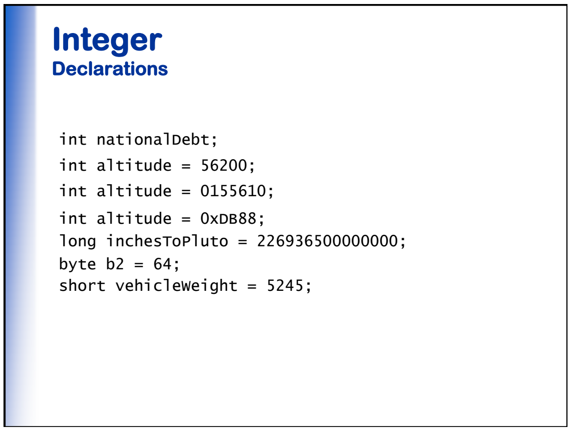
Integers
(cont.)

- byte
  - 8 bit value
    - -128 to 127
- short
  - 16 bit value
    - -32,768 to 32,767



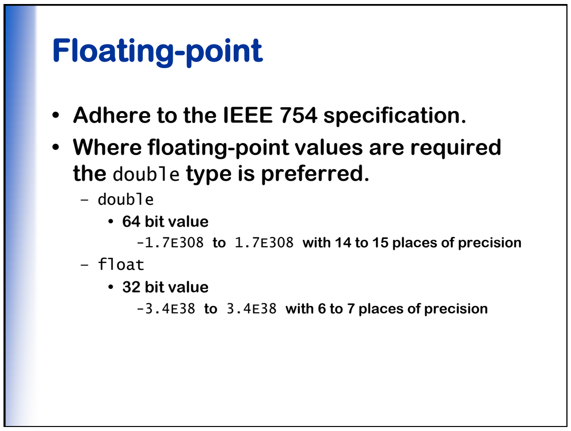
Integer
Literals

- Decimal literal, e.g. 64 or -256
- Octal notation, indicated by a leading zero, e.g. 0237
- Hexadecimal notation, indicated by a leading 0x or 0X, e.g. 0xD4 or 0Xd4
- long literals require an “L” or “l” suffix, the “L” is preferred



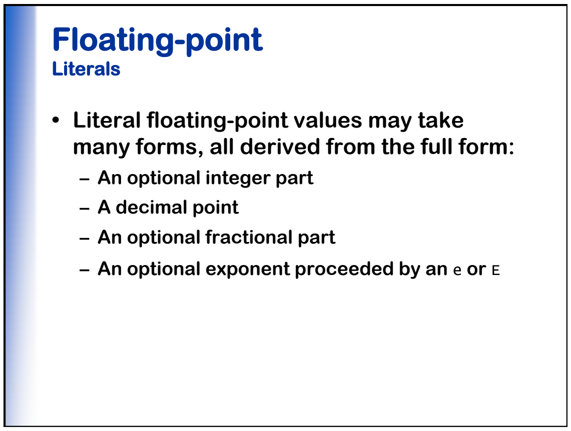
Integer
Declarations

```
int nationalDebt;
int altitude = 56200;
int altitude = 0155610;
int altitude = 0xDB88;
long inchesToPluto = 226936500000000;
byte b2 = 64;
short vehicleWeight = 5245;
```



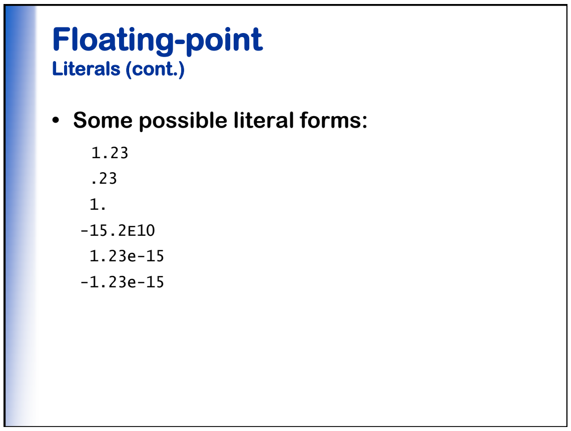
Floating-point

- Adhere to the IEEE 754 specification.
- Where floating-point values are required the `double` type is preferred.
  - `double`
    - 64 bit value
      `-1.7E308` to `1.7E308` with 14 to 15 places of precision
  - `float`
    - 32 bit value
      `-3.4E38` to `3.4E38` with 6 to 7 places of precision



Floating-point
Literals

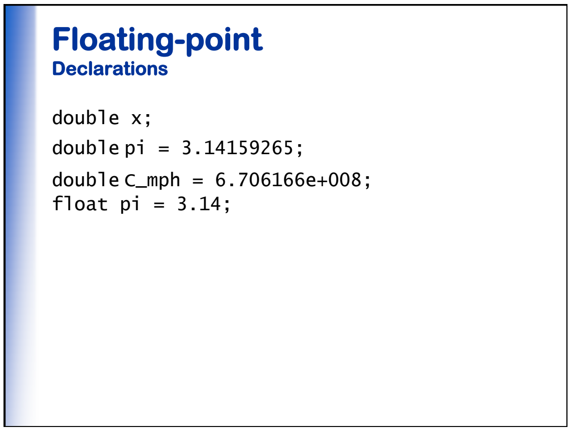
- Literal floating-point values may take many forms, all derived from the full form:
  - An optional integer part
  - A decimal point
  - An optional fractional part
  - An optional exponent proceeded by an `e` or `E`



Floating-point
Literals (cont.)

- Some possible literal forms:

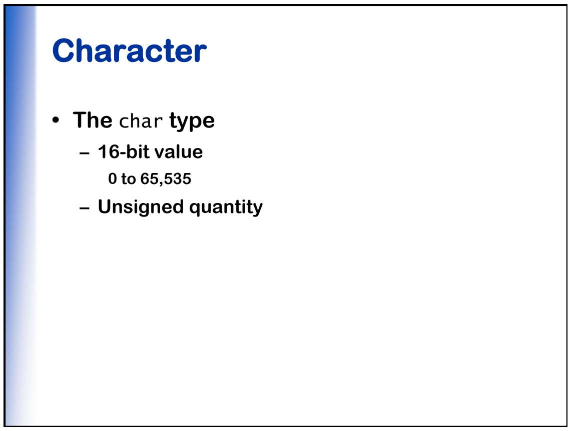
```
  1.23
  .23
  1.
-15.2E10
  1.23e-15
-1.23e-15
```



Floating-point

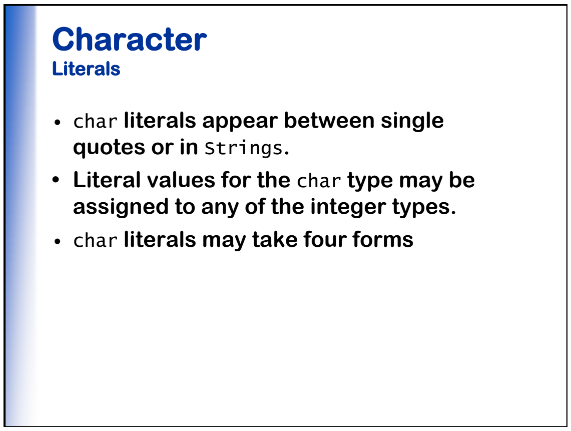
Declarations

```
double x;
double pi = 3.14159265;
double c_mph = 6.706166e+008;
float pi = 3.14;
```



Character

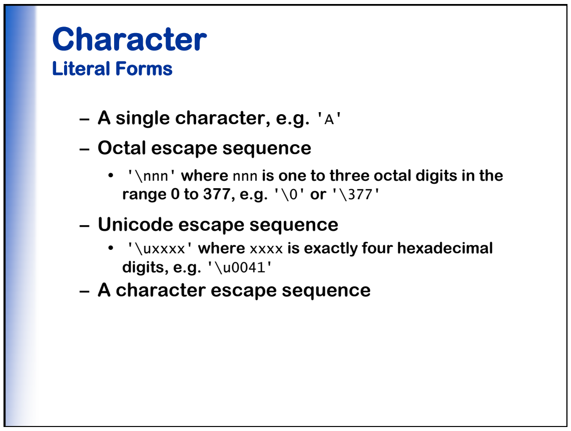
- The `char` type
  - 16-bit value
    - 0 to 65,535
  - Unsigned quantity



Character

Literals

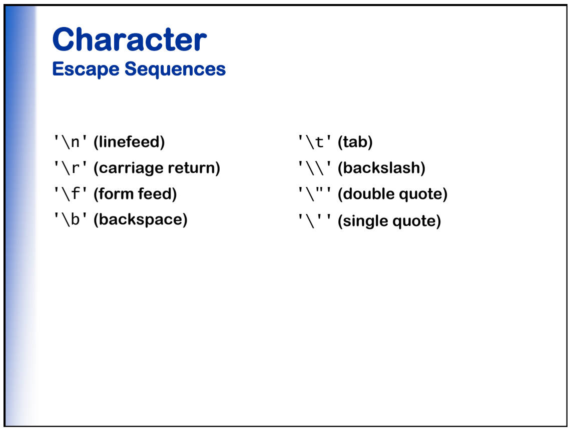
- `char` literals appear between single quotes or in `Strings`.
- Literal values for the `char` type may be assigned to any of the integer types.
- `char` literals may take four forms



Character

Literal Forms

- A single character, e.g. `'A'`
- Octal escape sequence
  - `'\nnn'` where `nnn` is one to three octal digits in the range 0 to 377, e.g. `'\0'` or `'\377'`
- Unicode escape sequence
  - `'\uxxxx'` where `xxxx` is exactly four hexadecimal digits, e.g. `'\u0041'`
- A character escape sequence

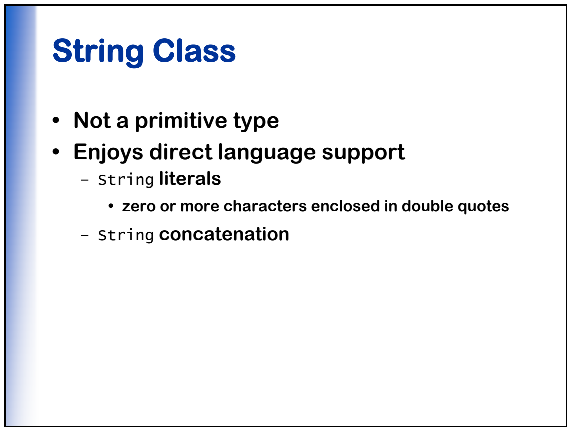


Character

Escape Sequences

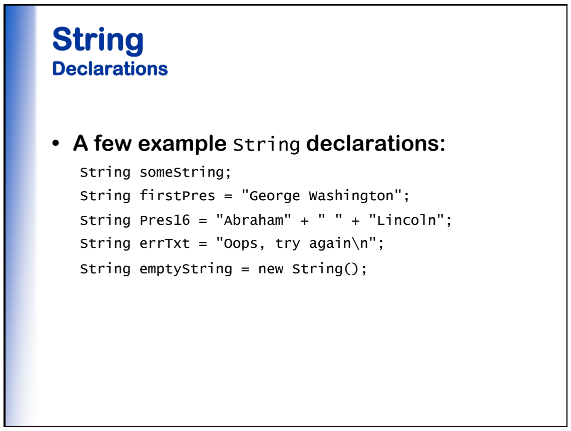
`'\n'` (linefeed)
`'\r'` (carriage return)
`'\f'` (form feed)
`'\b'` (backspace)

`'\t'` (tab)
`'\\'` (backslash)
`'\"'` (double quote)
`'\''` (single quote)



String Class

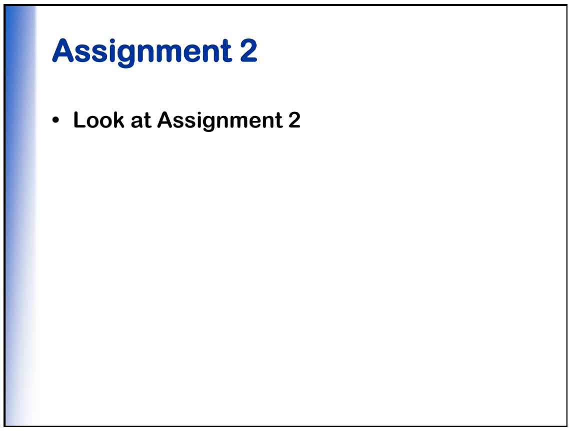
- Not a primitive type
- Enjoys direct language support
  - `String` literals
    - zero or more characters enclosed in double quotes
  - `String` concatenation



String
Declarations

- A few example `String` declarations:

```
String someString;
String firstPres = "George Washington";
String Pres16 = "Abraham" + " " + "Lincoln";
String errTxt = "Oops, try again\n";
String emptyString = new String();
```



Assignment 2

- Look at Assignment 2