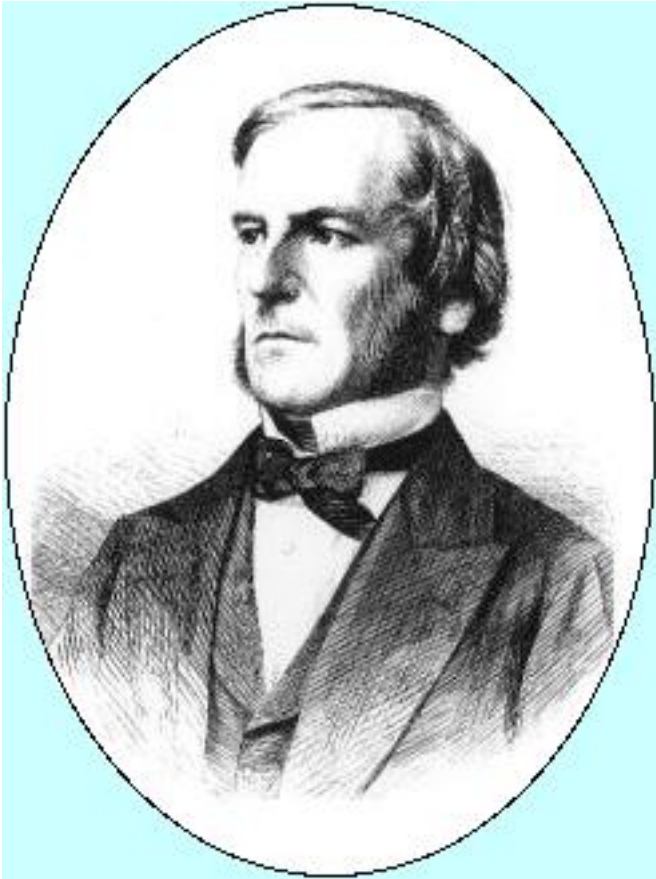


Decision and Repetition Statements

Statement Types in Java

- Programs in Java consist of a set of *classes*. Those classes contain *methods*, and each of those methods consists of a sequence of *statements*.
- Statements in Java fall into three basic types:
 - Simple statements
 - Compound statements
 - Control statements
- *Simple statements* are formed by adding a semicolon to the end of a Java expression.
- *Compound statements* (also called *blocks*) are sequences of statements enclosed in curly braces.
- *Control statements* fall into two categories:
 - *Conditional statements* that specify some kind of test
 - *Iterative statements* that specify repetition

Boolean Expressions



George Boole (1791–1871)

In many ways, the most important primitive type in Java is `boolean`, even though it is by far the simplest. The only values in the `boolean` domain are `true` and `false`, but these are exactly the values you need if you want your program to make decisions.

The name `boolean` comes from the English mathematician George Boole who in 1854 wrote a book entitled *An Investigation into the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*. That book introduced a system of logic that has come to be known as *Boolean algebra*, which is the foundation for the `boolean` data type.

Boolean Operators

- The operators used with the `boolean` data type fall into two categories: *relational operators* and *logical operators*.
- There are six relational operators that compare values of other types and produce a `boolean` result:

<code>==</code>	Equals	<code>!=</code>	Not equals
<code><</code>	Less than	<code><=</code>	Less than or equal to
<code>></code>	Greater than	<code>>=</code>	Greater than or equal to

For example, the expression `n <= 10` has the value `true` if `x` is less than or equal to 10 and the value `false` otherwise.

- There are also three logical operators:

<code>&&</code>	Logical AND	<code>p && q</code> means both <code>p</code> and <code>q</code>
<code> </code>	Logical OR	<code>p q</code> means either <code>p</code> or <code>q</code> (or both)
<code>!</code>	Logical NOT	<code>!p</code> means the opposite of <code>p</code>

Notes on the Boolean Operators

- Remember that Java uses `=` to denote assignment. To test whether two values are equal, you must use the `==` operator.
- It is not legal in Java to use more than one relational operator in a single comparison as is often done in mathematics. To express the idea embodied in the mathematical expression

$$0 \leq x \leq 9$$

you need to make both comparisons explicit, as in

$$0 \leq x \ \&\& \ x \leq 9$$

- The `||` operator means *either or both*, which is not always clear in the English interpretation of *or*.
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

Short-Circuit Evaluation

- Java evaluates the `&&` and `||` operators using a strategy called *short-circuit mode* in which it evaluates the right operand only if it needs to do so.
- For example, if `n` is 0, the right hand operand of `&&` in

`n != 0 && x % n == 0`

is not evaluated at all because `n != 0` is `false`. Because the expression

`false && anything`

is always `false`, the rest of the expression no longer matters.

- One of the advantages of short-circuit evaluation is that you can use `&&` and `||` to prevent execution errors. If `n` were 0 in the earlier example, evaluating `x % n` would cause a “division by zero” error.

The `if` Statement

The simplest of the control statements is the `if` statement, which occurs in two forms. You use the first form whenever you need to perform an operation only if a particular condition is true:

```
if (condition) {  
    statements to be executed if the condition is true  
}
```

You use the second form whenever you want to choose between two alternative paths, one for cases in which a condition is true and a second for cases in which that condition is false:

```
if (condition) {  
    statements to be executed if the condition is true  
} else {  
    statements to be executed if the condition is false  
}
```

Common Forms of the `if` Statement

The examples in the book use only the following forms of the `if` statement:

Single line `if` statement

```
if (condition) statement
```

Multiline `if` statement with curly braces

```
if (condition) {  
    statement  
    ... more statements ...  
}
```

`if/else` statement with curly braces

```
if (condition) {  
    statementstrue  
} else {  
    statementsfalse  
}
```

Cascading `if` statement

```
if (condition1) {  
    statements1  
} else if (condition2) {  
    statements2  
    ... more else/if conditions ...  
} else {  
    statementselse  
}
```


The ? : Operator

- In addition to the `if` statement, Java provides a more compact way to express conditional execution that can be extremely useful in certain situations. This feature is called the `? :` operator (pronounced *question-mark-colon*) and is part of the expression structure. The `? :` operator has the following form:

condition ? *expression*₁ : *expression*₂

- When Java evaluates the `? :` operator, it first determines the value of *condition*, which must be a `boolean`. If *condition* is `true`, Java evaluates *expression*₁ and uses that as the value; if *condition* is `false`, Java evaluates *expression*₂ instead.
- You could use the `? :` operator to assign the larger of `x` and `y` to the variable `max` like this:

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

The `switch` Statement

The `switch` statement provides a convenient syntax for choosing among a set of possible paths:

```
switch ( expression ) {  
    case  $v_1$ :  
        statements to be executed if expression =  $v_1$   
        break;  
    case  $v_2$ :  
        statements to be executed if expression =  $v_2$   
        break;  
    ... more case clauses if needed ...  
    default:  
        statements to be executed if no values match  
        break;  
}
```

Example of the `switch` Statement

The `switch` statement is useful when the program must choose among several cases, as in the following example:

```
public void run() {
    println("This program shows the number of days in a month.");
    int month = readInt("Enter numeric month (Jan=1): ");
    switch (month) {
        case 2:
            println("28 days (29 in leap years)");
            break;
        case 4: case 6: case 9: case 11:
            println("30 days");
            break;
        case 1: case 3: case 5: case 7: case 8: case 12:
            println("31 days");
            break;
        default:
            println("Illegal month number");
            break;
    }
}
```

The `while` Statement

The `while` statement is the simplest of Java's iterative control statements and has the following form:

```
while ( condition ) {  
    statements to be repeated  
}
```

When Java encounters a `while` statement, it begins by evaluating the condition in parentheses, which must have a boolean value.

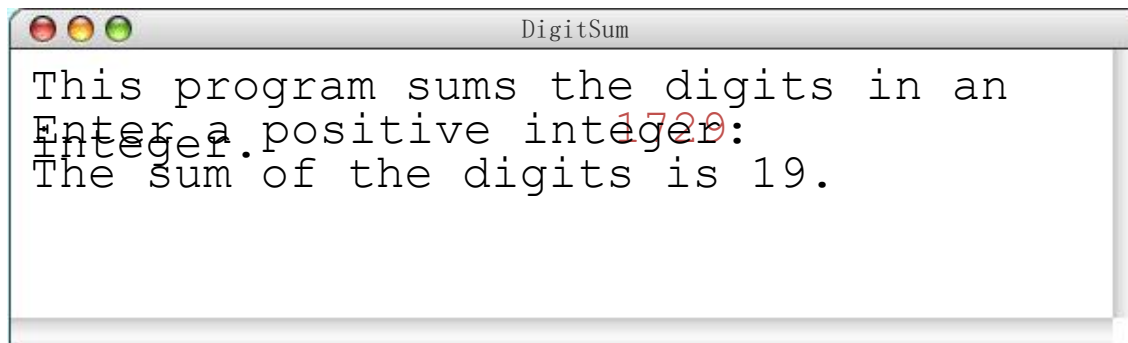
If the value of *condition* is `true`, Java executes the statements in the body of the loop.

At the end of each cycle, Java reevaluates *condition* to see whether its value has changed. If *condition* evaluates to `false`, Java exits from the loop and continues with the statement following the closing brace at the end of the `while` body.

The DigitSum Program

```
public void run() {  
    println("This program sums the digits in an integer.");  
    int n = readInt("Enter a positive integer: ");  
    int dsum = 0;  
    while (n > 0) {  
        dsum += n % 10;  
        n /= 10;  
    }  
    println("The sum of the digits is " + dsum);  
}
```

n	dsum
0	19



do...while loop

- The do loop is just like a while loop, except that do executes a given statement or block until a condition is false.
- The main difference is that
 - while loops test the condition before looping, making it possible that the body of the loop will never execute if the condition is false the first time it's tested.
- do loops run the body of the loop at least once before testing the condition

do...while loop

- do loops look like this:

```
do {
```

```
bodyOfLoop;
```

```
} while (condition);
```

while vs. do...while loop

```
int counter = 1;
while (counter < 10) {
    System.out.println("Hello");
    counter++;
}
```

```
do {
    System.out.println("Hello");
    counter++;
} while (counter < 10);
```


The `for` Statement

The `for` statement in Java is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for ( init ; test ; step ) {  
    statements to be repeated  
}
```

Java evaluates a `for` statement by executing the following steps:

1. Evaluate *init*, which typically declares a *control variable*.
2. Evaluate *test* and exit from the loop if the value is `false`.
3. Execute the statements in the body of the loop.
4. Evaluate *step*, which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing `for` and `while`

The `for` statement

```
for ( init ; test ; step ) {  
    statements to be repeated  
}
```

is functionally equivalent to the following code using `while`:

```
init;  
while ( test ) {  
    statements to be repeated  
    step;  
}
```

The advantage of the `for` statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Exercise: Reading `for` Statements

Describe the effect of each of the following `for` statements:

1. `for (int i = 1; i <= 10; i++)`

This statement executes the loop body ten times, with the control variable `i` taking on each successive value between 1 and 10.

2. `for (int i = 0; i < N; i++)`

This statement executes the loop body `N` times, with `i` counting from 0 to `N - 1`. This version is the standard Repeat-N-Times idiom.

3. `for (int n = 99; n >= 1; n -= 2)`

This statement counts backward from 99 to 1 by twos.

4. `for (int x = 1; x <= 1024; x *= 2)`

This statement executes the loop body with the variable `x` taking on successive powers of two from 1 up to 1024.

The Checkerboard Program

```
public void run() {  
    double sqSize = (double) getHeight() / N_ROWS;  
    for (int i = 0; i < N_ROWS; i++) {  
        for (int j = 0; j < N_COLUMNS; j++) {  
            double x = j * sqSize;  
            double y = i * sqSize;  
            GRect sq = new GRect(x, y, sqSize, sqSize);  
            sq.setFilled((i + j) % 2 != 0);  
            add(sq);  
        }  
    }  
}
```

sqSize

30.0

i

8

j

8

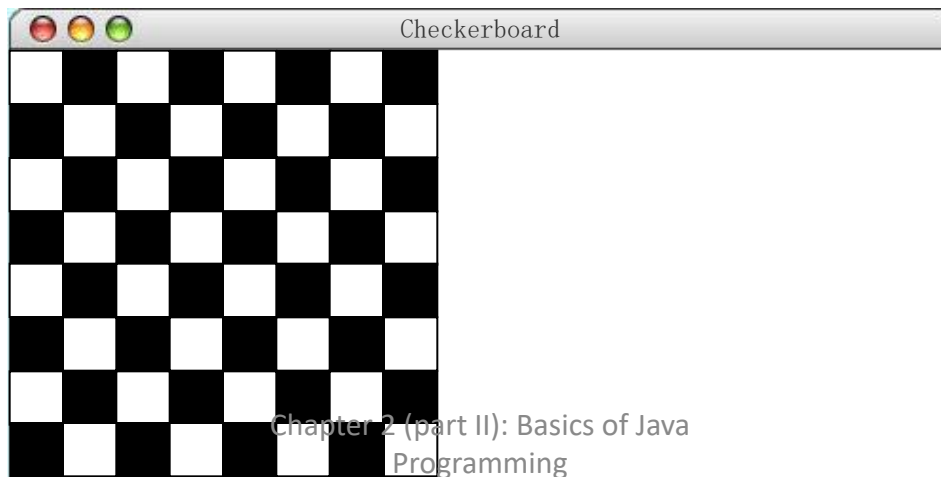
x

210.0

y

210.0

sq



Simple Graphical Animation

The `while` and `for` statements make it possible to implement simple graphical animation. The basic strategy is to create a set of graphical objects and then execute the following loop:

```
for (int i = 0; i < N_STEPS; i++) {  
    update the graphical objects by a small amount  
    pause(PAUSE_TIME) ;  
}
```

On each cycle of the loop, this pattern updates each animated object by moving it slightly or changing some other property of the object, such as its color. Each cycle is called a *time step*.

After each time step, the animation pattern calls `pause`, which delays the program for some number of milliseconds (expressed here as the constant `PAUSE_TIME`). Without the call to `pause`, the program would finish faster than the human eye can follow.

The AnimatedSquare Program

```
public void run() {  
    GRect square = new GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);  
    square.setFilled(true);  
    square.setFillColor(Color.RED);  
    add(square);  
    double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;  
    double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;  
    for (int i = 0; i < N_STEPS; i++) {  
        square.move(dx, dy);  
        pause(PAUSE_TIME);  
    }  
}
```

i

101

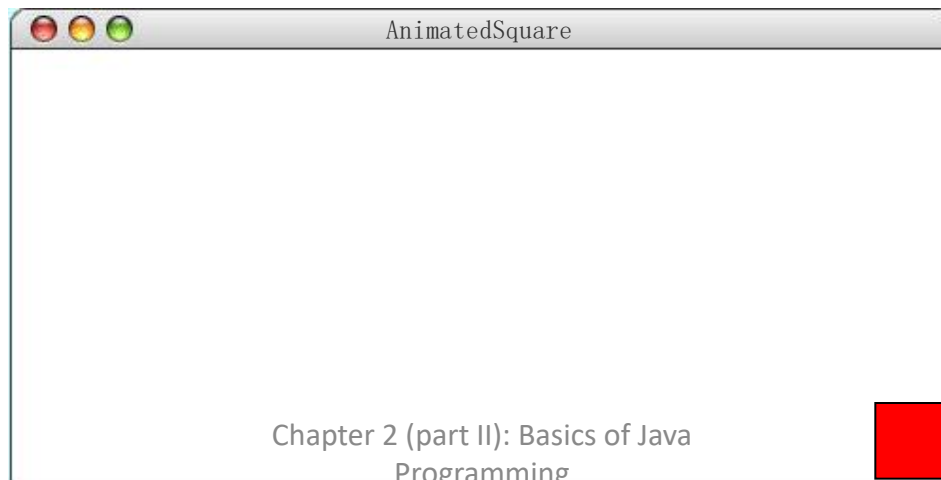
dx

3.0

dy

1.7

square



Java Standard input, output and error streams

Stream

- A stream is a flowing sequence of characters
- Standard input stream: a stream that reads characters from the keyboard
 - a convenient place for an old-fashioned text-based application to get input from the user
- Standard output stream: a stream that writes its contents to the display
 - A convenient place for an old-fashioned text-based application to display its output
- Standard error stream: a stream that displays error messages to the user

Java Standard Input Stream

- The most common standard input device is keyboard.
- **java.lang.System** class contain a standard input object called **in**
 - **System.in** is, thus, a standard input in java
 - **System.in.read** method reads a single character and returns either the character that was read or, if there are no more characters to be read, -1
 - **System.in** is a class variable that is a reference to an object implementing the standard input stream

```
System.out.println("Enter a character");  
int ch= System.in.read();  
System.out.println("You entered "+(char)ch);
```

System.in.read() method

- public abstract int **read()** throws **IOException**
- reads the next byte of data from the input stream.
- The value byte is returned as an **int** in the range 0 to 255.
- If no byte is available because the end of the stream has been reached, the value -1 is returned.

System.in.read() method...

- This method blocks until
 - input data is available
 - the end of the stream is detected, or
 - an exception is thrown
- A subclass must provide an implementation of this method.
- It returns:
 - the next byte of data, or
 - -1 if the end of the stream is reached.
- It throws: **IOException** if an I/O error occurs.

System.in.read() method...

```
class StandardInputer {  
  
    public static void main(String[] a) throws IOException {  
        System.out.print("Please enter a sequence of characters and press Enter key: ");  
        int count = 0;  
        int readByte;  
        while ((readByte = System.in.read()) != 10) { // 10 is the ASCII value of LF (Line Feed) character,  
            // that is when one press the Enter key of the keyboard. Please check ASCII value of LF from ASCII tables  
            count++;  
            System.out.println(count + " th character is " + (char) readByte + ": " + readByte + "(ASCII value)");  
            // (char) is used to cast ASCII values  
            // to the corresponding char  
        }  
        System.out.println("==>Your input has " + count + " characters.");  
    }  
}
```

Java Standard Input Stream...

- How could we take more than a character input from keyboard in java language?
 - java.util.Scanner class is one option to implement java input from the keyboard

```
Scanner scan = new Scanner(System.in);  
System.out.println("What is your name?");  
String name = scan.next();  
System.out.println("Your name is "+name);
```

Java Standard Output Stream

- The most common standard output device is screen/monitor. `System.out.println("Welcome to`
- **java.lang.System** class contain a standard output object called **out**
 - **System.out** is, thus, a standard output in java
 - System.out is a class variable that is a reference to an object implementing the standard output stream
- How could we display output to console display?
 - Using **print** or **println** method of the **System.out** object

Java Standard Error stream

- To display error messages to the user
- `System.err`
- implements the standard error stream

```
System.out.println("What is your name?");
```

```
String name = scan.next();
```

```
System.out.println("Your name is " + name);
```

```
if (name.length() < 3) {
```

```
    System.err.println("Opps, Ethiopian Name can not be less than three characters long");
```

packages, interfaces, classes, objects and methods in java

Packages in Java

- Packages are Java's way of doing large-scale design and organization.
- They are used to categorize and group classes
- You first declare the name of the package by using a *package* statement.
- Then you define a class, just as you would normally do.
- That class, and any other classes also declared inside this same package name, are grouped together

Packages in java...

- Syntax: *package* <package name here>;
- Example:

```
package myFirstPackage;  
public class MyPublicClass  
{  
    . . .  
}
```

Packages in java...

- If a package statement appears in a Java source file, it must be the first thing in that file (except for comments and white space, of course)
- Packages can be further organized into a hierarchy somewhat analogous to the inheritance hierarchy
 - where each “level” usually represents a smaller, more specific grouping of classes

Packages in java...

- Suppose you want to use a lot of classes from a package, a package with a long name, or both.
- You don't want to have to refer to your classes as **that.really.long.package.name.ClassName**.
- Java allows you to “import” the names of those classes (public ones) from a package into your program
 - Then, you need not write the complete class name in your java code
 - You could write just the class name

Packages in java...

- What if you want to use/import several classes from that same package?
- Instead of importing each class of the package separately, we can use `*` after the packages
- Example: Instead of :
 - `that.really.long.package.name.ClassName1`
 - `that.really.long.package.name.ClassName2`
 - `that.really.long.package.name.ClassName3`
- One can issue **`that.really.long.package.name.*`**

Interfaces

- provide templates of behavior that other classes are expected to implement
 - Contain method descriptions and/or static variables
 - Methods are by default *public* and *abstract*
 - Variables are *public*, *static* and *final*
- declared in source files and also are compiled into .class files
- Syntax: *public interface* [interface name]{methods, variables}

Interfaces...

- hierarchy of interfaces is not limited to a single superclass
 - so they allow a form of multiple-inheritance
- Interfaces could extend other interfaces
- Classes implement interfaces

classes

- Class is a concept that encapsulates methods and attributes
- Definition of a class is as follow:
 - `class MyClassName {}`
 - `class myClassName extends mySuperClassName {}` <- for a class that inherits another class
 - `class MyRunnableClassName implements interface(s) {}` <- for a class that implements an interface

Classes...

- Abstract classes are classes that can not be instantiated
 - Is marked abstract
 - May or may not contain abstract methods
- **java.util.Calendar** is an abstract class and we can not instantiate it whereas **java.util.GregorianCalendar** is a concrete class that could be instantiated
 - **GregorianCalendar** is subclass of **Calendar**

Classes...

```
public class StandardInputer extends java.lang.Object {
```

Calendar is abstract; cannot be instantiated

....

(Alt-Enter shows hints)

```
static public void main(String[] a) throws IOException {
```

```
    Calendar cl= new Calendar(); //Calendar is abstrcat; cannot be instantiated
```

```
    GregorianCalendar gc = new GregorianCalendar();
```

```
    System.out.println("Today's date/time: " + gc.getTime());
```

Classes...

- Classes that are marked as *final* can not be subclassed
- Example: `java.lang.Boolean` is a final class and we can not create a class that extends this class

Objects (instances of classes)

- Classes can be instantiated using the *new* keyword
- The appropriate constructor is invoked depending on the setup of the class (constructor is discussed in the next subsection)
- Example: `MyClassName ob1 = new MyClassName ();`
- **ob1** is an object of the class `MyClassName`

Objects (instances of classes)...

- Objects could access their respective attributes and methods using the dot (.) operator
- For example, assume **attr1** is an attribute of `MyClassName` class and **method1** is a method of the same class.
- Thus, **ob1** can access its elements as follow:
 - **ob1.attr1**
 - **ob1.method1()**

Attributes of a java class

- Attributes of a java class could be generally categorized as **static** fields or **instance** fields
- Static attributes are associated with the respective class of which value could be accessed using just the class name
 - These fields are not specific to a certain instantiated object though objects could also access such fields
- Instance variables (also called non-static fields) are attributes that are associated with instantiated objects

Attributes of a java class...

- Instance variables could not be accessed just by the class name as they demand instantiation of the class
- Attributes could have visibility of
 - public: globally accessible
 - Private: accessible by the methods inside the class
 - Protected: accessed by the class and any class that inherits the class
 - Package(default): accessible by classes that are inside the same package of the class that contain the attribute

Attributes of a java class...

- To declare constants in Java, use final variables:
 - public static **final** int aConstantInt = 123;
 - public **final** String aConstantString = “Hello world!”;

methods

- classes and objects provide the framework, and class and instance variables provide a way of holding that class or object's attributes and state,
- Methods provide an object's behavior and define how that object interacts with other objects in the system
- The **method's *signature*** is a combination of
 - *the name of the method*
 - *the type of object or* base type this method returns, and
 - a list of parameters.

Methods...

- Sometimes, in the body of a method definition, you may want to refer to the current object, for example, to refer to that object's instance variables or to pass the current object as an argument to another method.
- **this** is a reference to the current instance of a class

Methods...

- Abstract methods are methods that are without body , that is, don't contain method implementation
 - An abstract class could contain abstract method
 - All methods of an interface are abstract by default
- Methods that can not be overridden are those methods which are marked *final* and subclasses of the class that contain the method can not override the method

(Access) Modifiers

- *Modifiers are prefixes that can be applied in various combinations to the methods and variables within a class and, some, to the class itself*
- There is a long and varied list of modifiers
- The order of modifiers is irrelevant to their meaning
 - your order can vary and is really a matter of taste.
 - Pick a style and then be consistent with it throughout all your classes

Access Modifiers...

- Here is the recommended order:
- <access> static abstract synchronized
<unusual> final native where
 - <access> can be public, protected, or private, and
 - <unusual> includes volatile and transient.
- All the modifiers are essentially optional

Constructors

Constructors

- **Constructors in Java are very similar to C++**
- **You can overload constructors (like any other method)**
- **A constructor which doesn't get any parameter is called "empty constructor"**
- **You may prefer not to have a constructor at all, in which case it is said that you have by default an "empty constructor"**
- **A constructor can call another constructor of the same class using the 'this' keyword**
- **Calling another constructor can be done only as the first instruction of the calling constructor**

Examples in following slides...

Constructors

Example 1:

```
public class Person {  
    String name = ""; // fields can be initialized!  
    Date birthDate = new Date();  
    public Person() {} // empty constructor  
    public Person(String name, Date birthDate) {  
        this(name); // must be first instruction  
        this.birthDate = birthDate;  
    }  
    public Person(String name) {  
        this.name = name;  
    }  
}
```


Constructors

Example 2:

```
public class Person {  
    String name = "";  
    Date birthDate = new Date();  
    public Person(String name, Date birthDate) {  
        this.name = name;  
        this.birthDate = birthDate;  
    }  
}
```

```
Person p; // OK
```

```
p = new Person(); // not good - compilation error
```

Constructors

- **constructor:** Initializes the state of new objects.

```
public type(parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- no return type is specified; implicitly "returns" the new object

```
public class Point {  
    private int x;  
    private int y;  
    public Point(int initialX, int  
initialY) {  
        x = initialX;  
        y = initialY;  
    }  
}
```

Multiple constructors

- A class can have multiple constructors.
 - Each one must accept a unique set of parameters.
- *Example:* A `Point` constructor with no parameters that initializes the point to (0, 0).

// Constructs a new point at (0, 0) .

```
public Point() {  
    x = 0;  
    y = 0;  
}
```

The keyword `this`

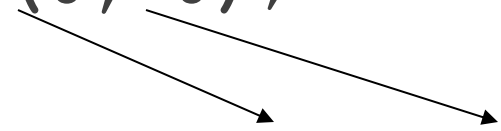
- **`this`** : Refers to the implicit parameter inside your class.

(a variable that stores the object on which a method is called)

- Refer to a field: `this . field`
- Call a method: `this . method (parameters) ;`
- One constructor can call another: `this (parameters) ;`

Calling another constructor

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0);  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```



- Avoids redundancy between constructors