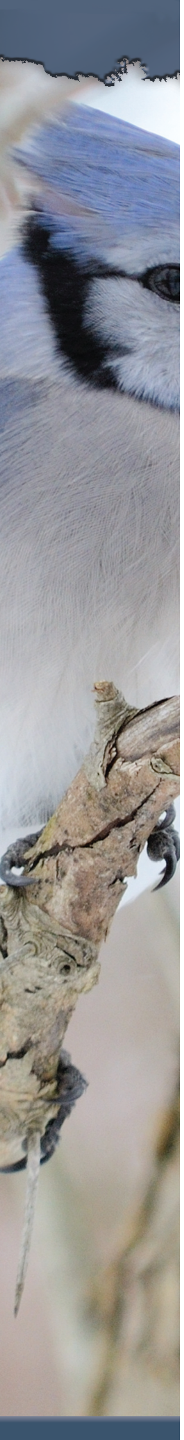
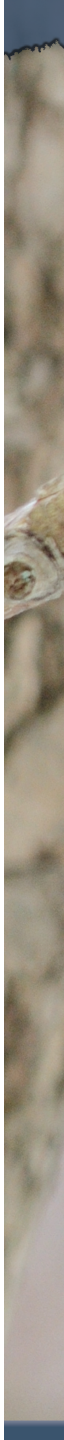
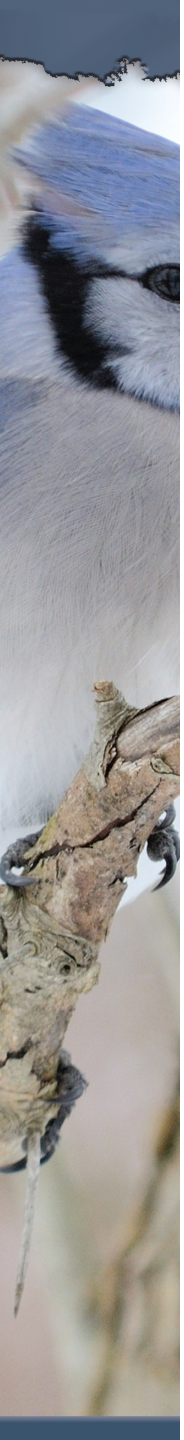


More-Sophisticated Behavior

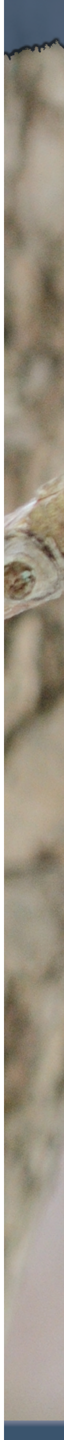
Using library classes to implement some more advanced functionality





Main concepts to be covered

- Using library classes
- Reading documentation



The Java class library

- Thousands of classes.
- Tens of thousands of methods.
- Many useful classes that make life much easier.
- Library classes are often inter-related.
- Arranged into packages.

Working with the library

- A competent Java programmer must be able to work with the libraries.
- You should:
  - know some important classes by name;
  - know how to find out about other classes.
- Remember:
  - we only need to know the *interface*, not the *implementation*.

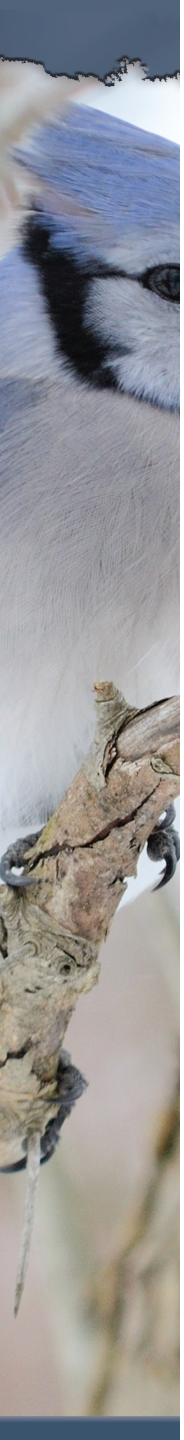
A Technical Support System

- A textual, interactive dialog system.
- Idea based on '*Eliza*' by Joseph Weizenbaum (MIT, 1960s).
- Explore *tech-support-complete* …
- … The program appears to respond intelligently to the user's typed input.
- Explore *tech-support1* – an incomplete version.

A Technical Support System

Paleo-AI

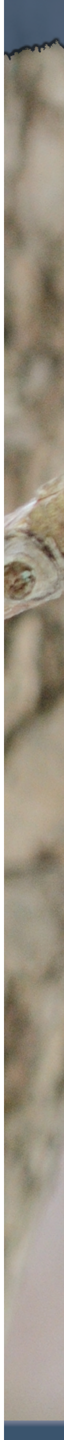
- A textual, interactive dialog system.
- Idea based on '*Eliza*' by Joseph Weizenbaum (MIT, 1960s).
- Explore *tech-support-complete* …
- … The program appears to respond intelligently to the user's typed input.
- Explore *tech-support1* – an incomplete version.



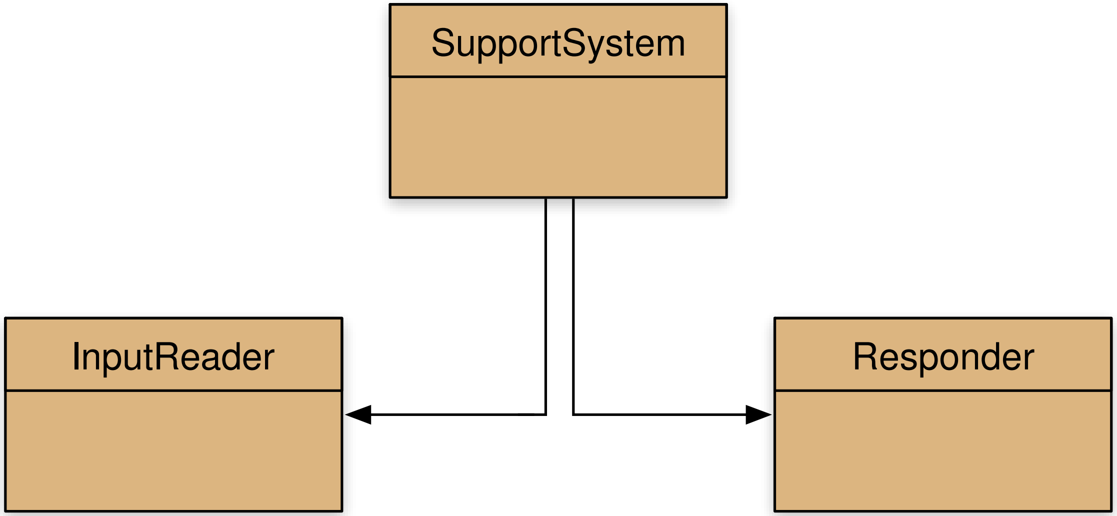
Main loop structure

```
boolean finished = false;

while (!finished) {

     do something

     if (exit condition) {
          finished = true;
     } else {
          do something more
     }
}
```

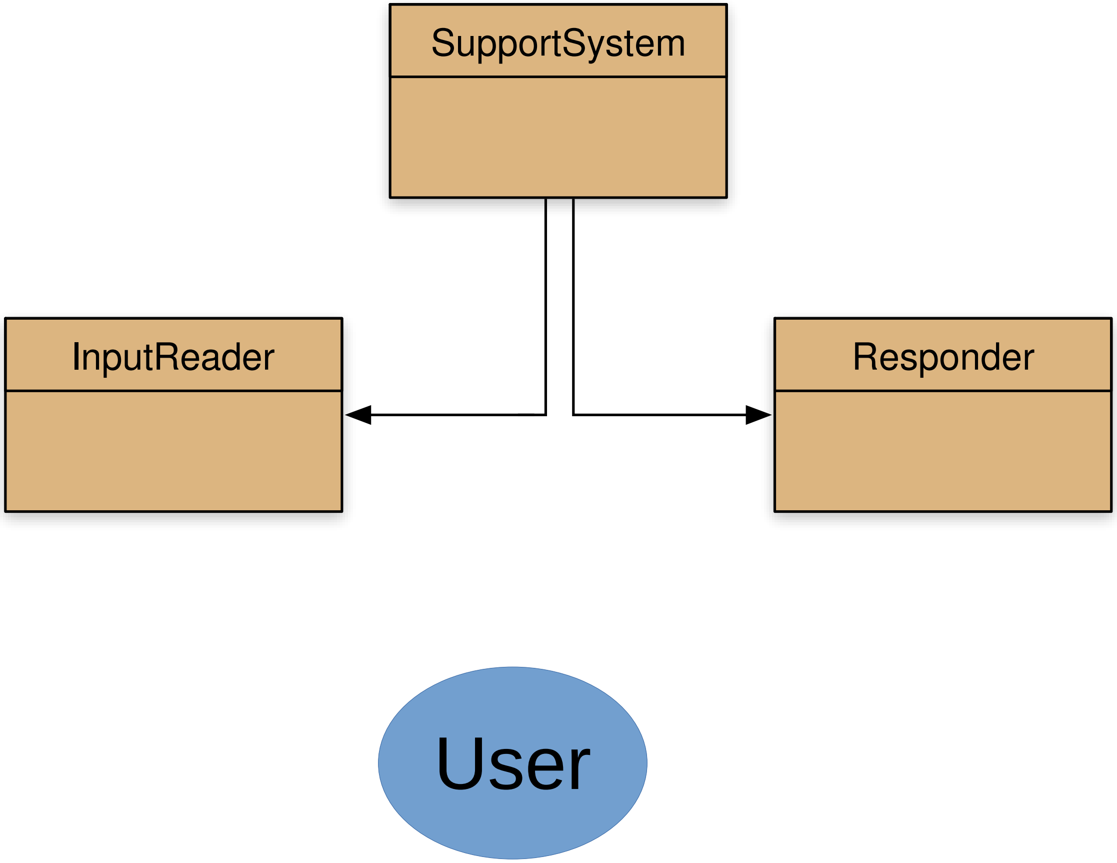
A common iteration pattern.



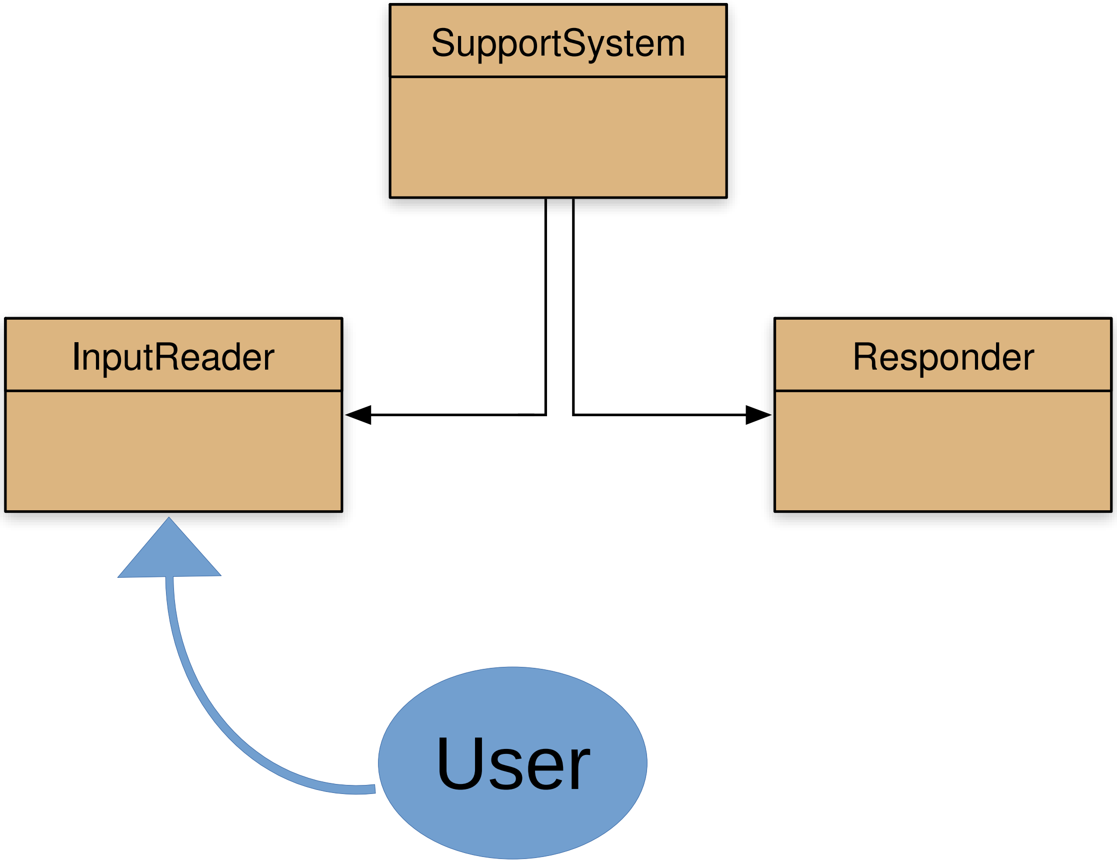
Modularization



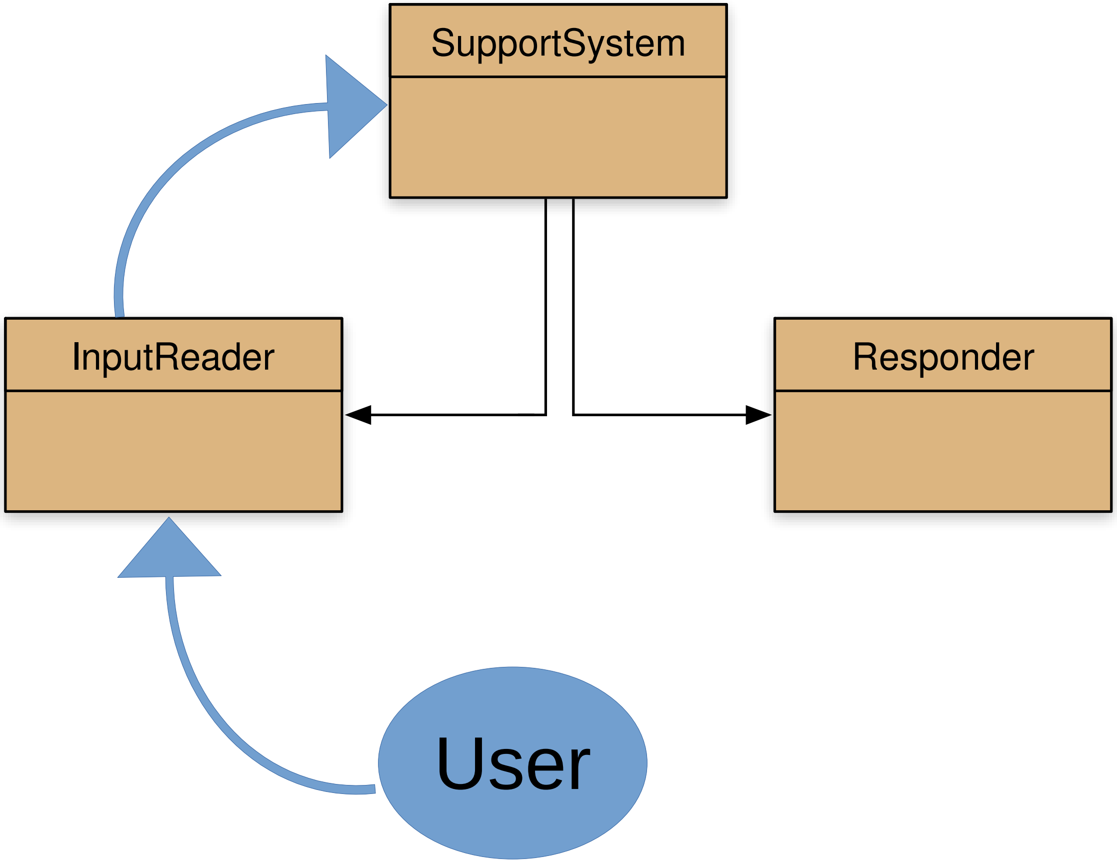
Modularization



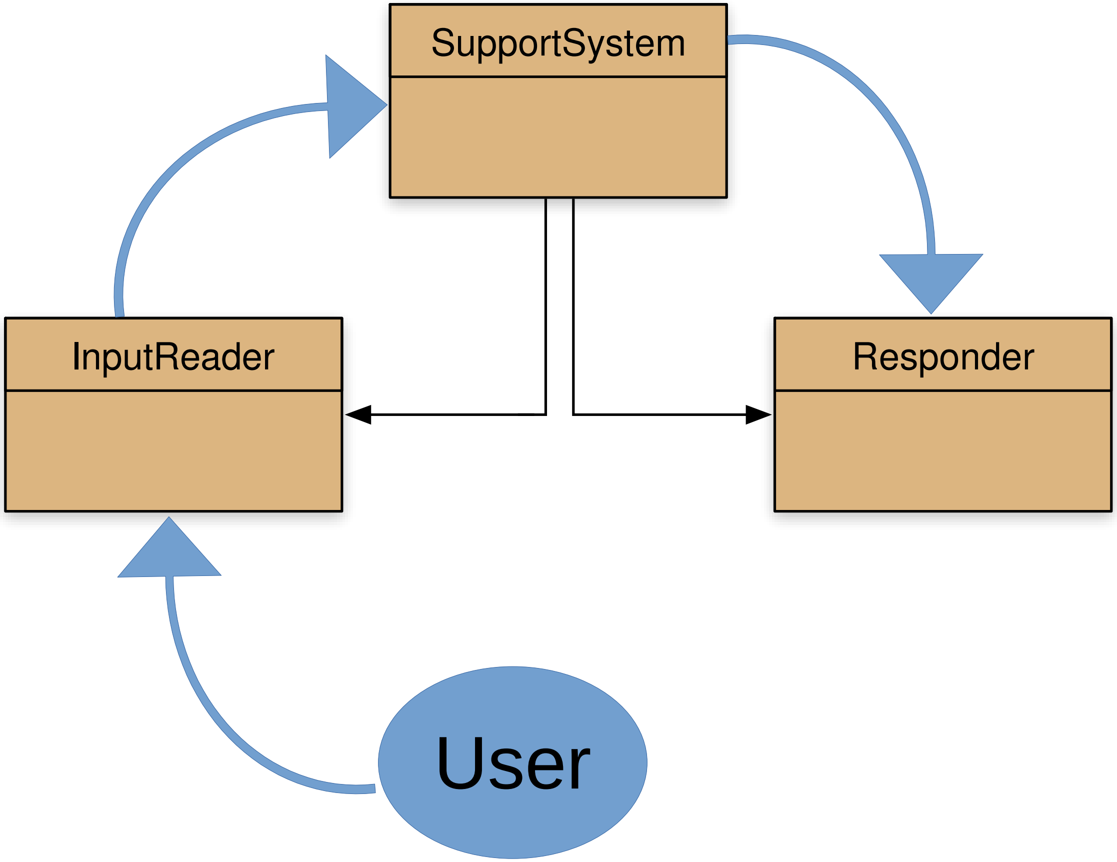
Modularization



Modularization

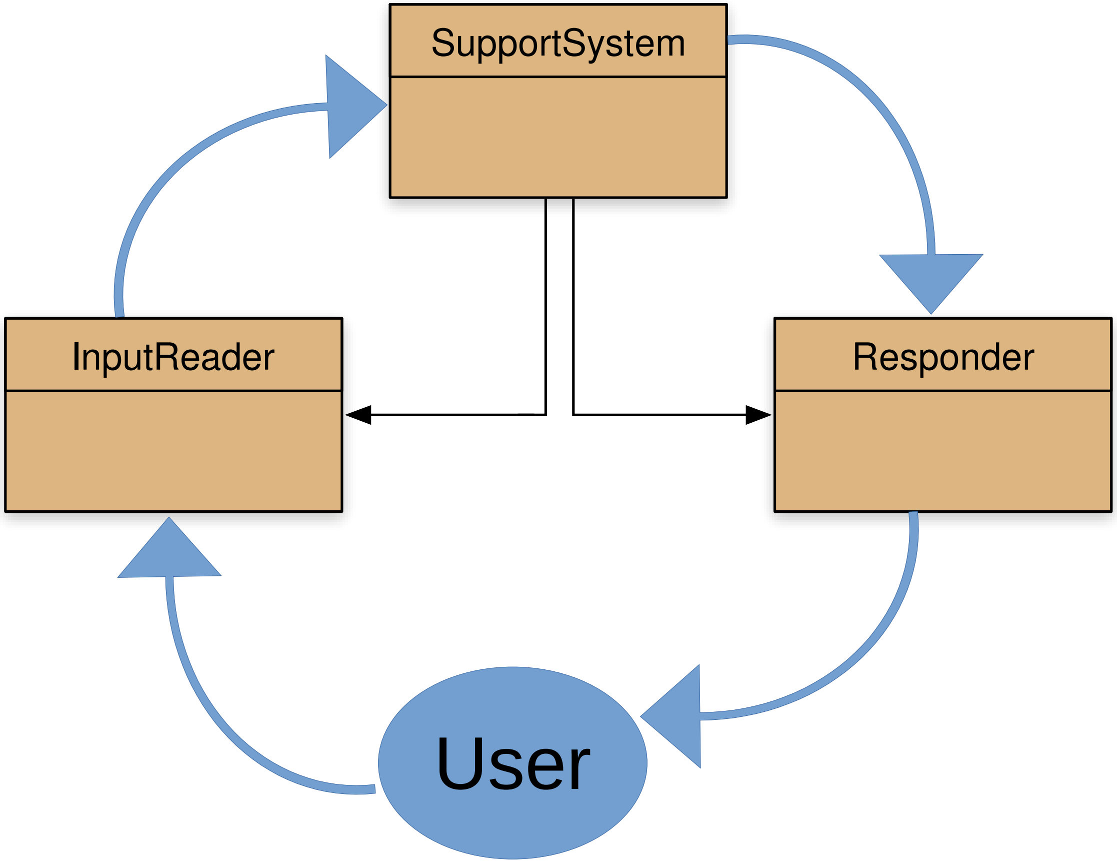


Modularization



© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

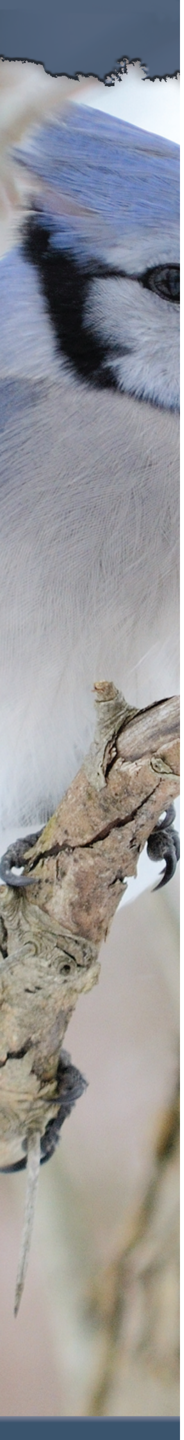
Modularization



Main loop body in SupportSystem

```
String input = reader.getInput();
...
String response = responder.generateResponse();
System.out.println(response);
```

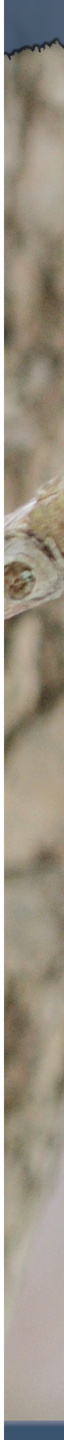
NB: `input` is ignored by the `Responder` in this version



The exit condition

```
String input = reader.getInput();

if (input.startsWith("bye")) {
     finished = true;
}
```

- Where does ‘**`startsWith`**’ come from?
- What is it? What does it do?
- How can we find out?

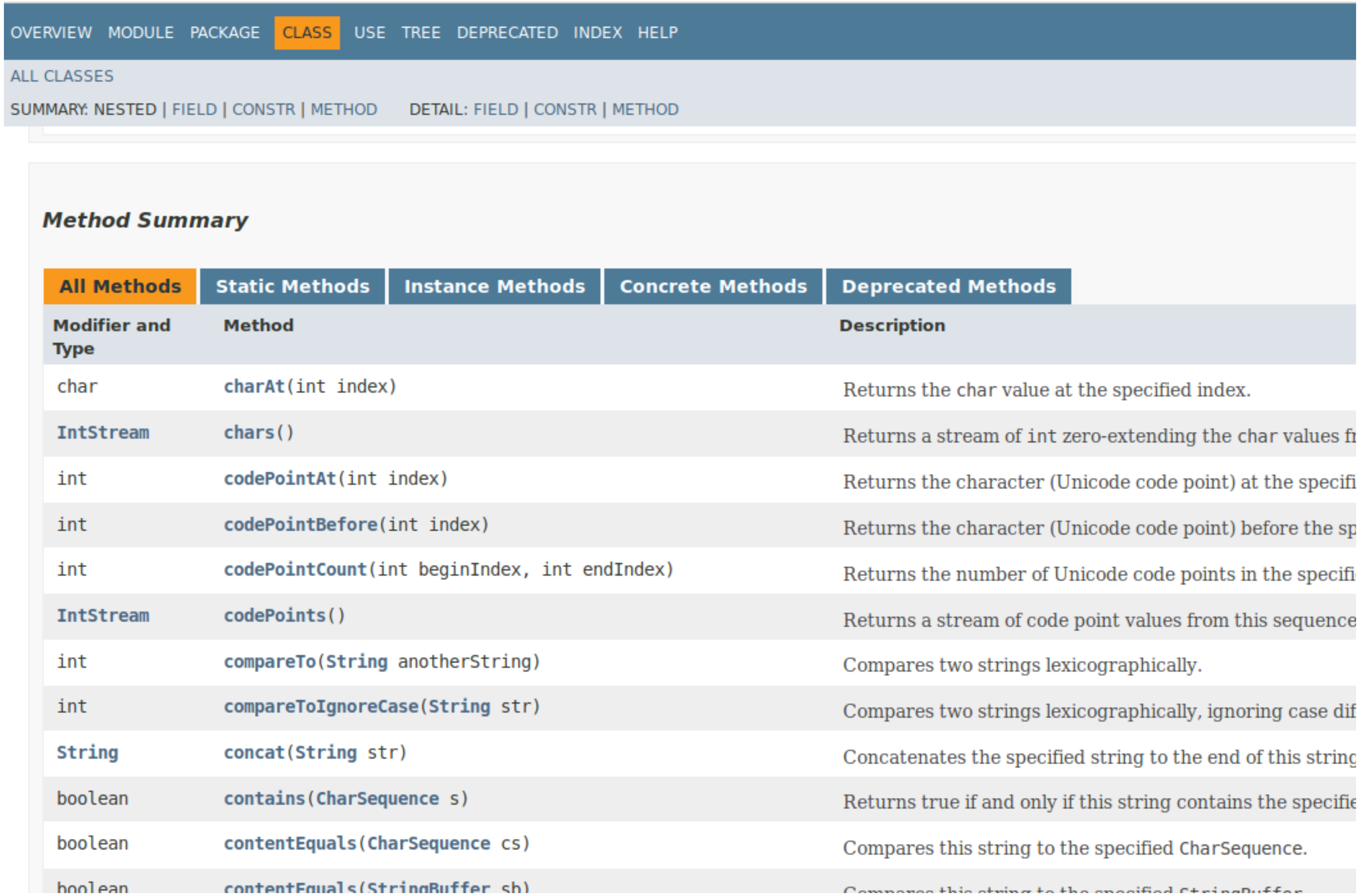


Reading class documentation

- Documentation of the Java libraries in HTML format;
- Readable in a web browser
- Class API: *Application Programmers' Interface*
- Interface description for all library classes

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/module-summary.html

API Reference


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
All Methods
Static Methods
Instance Methods
Concrete Methods
Deprecated Methods

Method Summary

Modifier and Type	Method	Description
char	**charAt**(int index)	Returns the char value at the specified index.
IntStream	**chars**()	Returns a stream of int zero-extending the char values fr
int	**codePointAt**(int index)	Returns the character (Unicode code point) at the specifi
int	**codePointBefore**(int index)	Returns the character (Unicode code point) before the sp
int	**codePointCount**(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specifi
IntStream	**codePoints**()	Returns a stream of code point values from this sequence
int	**compareTo**(**String** anotherString)	Compares two strings lexicographically.
int	**compareToIgnoreCase**(**String** str)	Compares two strings lexicographically, ignoring case dif
String	**concat**(**String** str)	Concatenates the specified string to the end of this strinc
boolean	**contains**(**CharSequence** s)	Returns true if and only if this string contains the specifie
boolean	**contentEquals**(**CharSequence** cs)	Compares this string to the specified CharSequence.
boolean	**contentEquals**(**StringBuffer** sb)	Compares this string to the specified StringBuffer

Interface vs implementation

The documentation includes

- the name of the class;
- a general description of the class;
- a list of constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method

→ **the *interface* of the class**

Interface vs implementation

The documentation includes

- the name of the class;
- a general description of the class;
- a list of constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method

Important

→ **the *interface* of the class**

Interface vs implementation

*The documentation **does not** include*

- private fields (most fields are private)
- private methods
- the bodies (source code) of methods

→ **the *implementation* of the class**

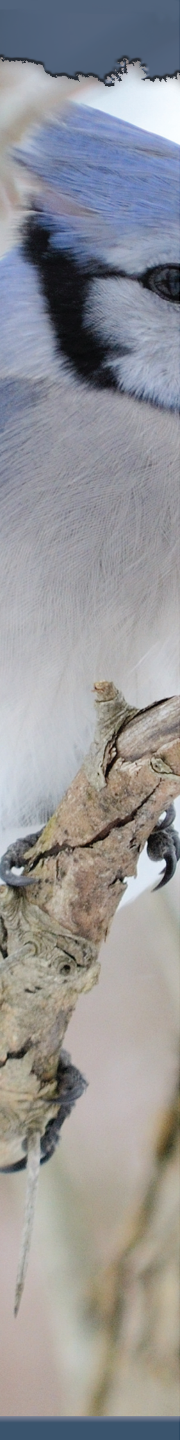
Interface vs implementation

*The documentation **does not** include*

- private fields (most fields are private)
- private methods
- the bodies (source code) of methods

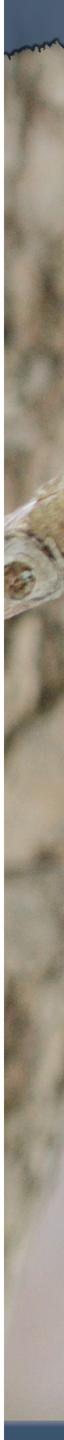
→ **the *implementation* of the class**

Not so much



Documentation for startsWith

- **startsWith**
  - **public boolean startsWith(String prefix)**
- Tests if this string starts with the specified prefix.
- Parameters:
  - **prefix** - the prefix.
- Returns:
  - **true** if the …; **false** otherwise

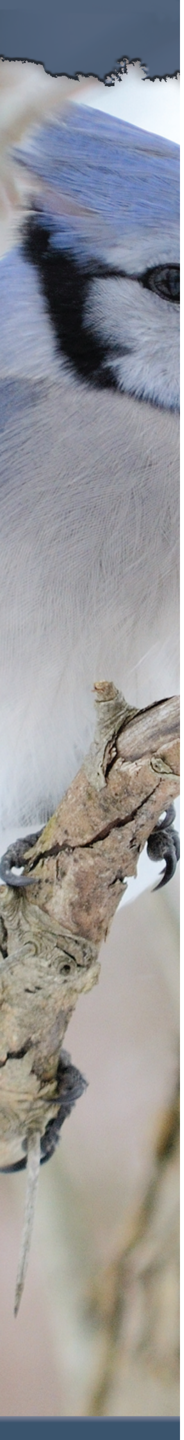


Methods from **`String`**

- **`contains`**
- **`endsWith`**
- **`indexOf`**
- **`substring`**
- **`toUpperCase`**
- **`trim`**
- Beware: strings are *immutable*!

Using library classes

- Classes organized into packages.
- Classes from the library must be *imported* using an **`import`** statement;
  - except from the **`java.lang`** package.
- They can then be used like classes from the current project.



Packages and import

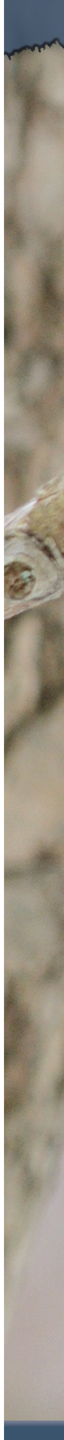
- Single classes may be imported:

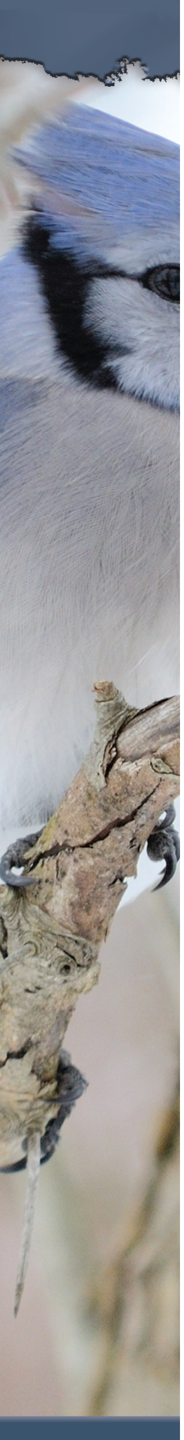
```
import java.util.ArrayList;
```

- All classes from a package can be imported (considered bad style):

```
import java.util.*;
```

- Importation does not involve source code insertion.

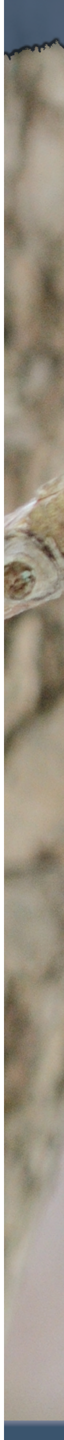




Adding random behavior

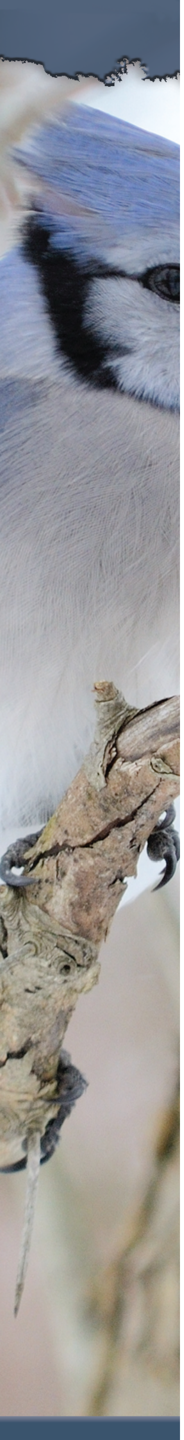
- The library class **`Random`** can be used to generate random numbers:

```
import java.util.Random;
...
Random rand = new Random();
...
int num = rand.nextInt();
int value = rand.nextInt(100);
int index = rand.nextInt(list.size());
```



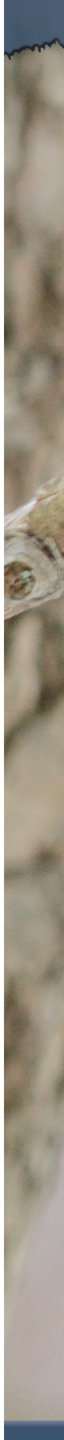
Selecting random responses

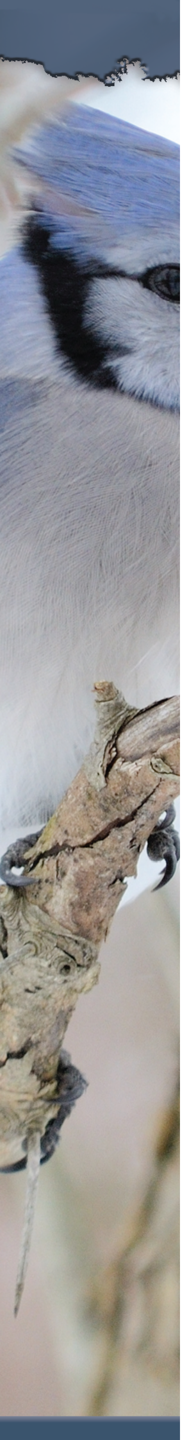
```
public Responder() {
   randomGenerator = new Random();
   responses = new ArrayList<>();
   fillResponses();
}

public void fillResponses() {
   fill responses with a selection of response strings
}

public String generateResponse() {
   int index = randomGenerator.nextInt(responses.size());
   return responses.get(index);
}
```



Parameterized classes

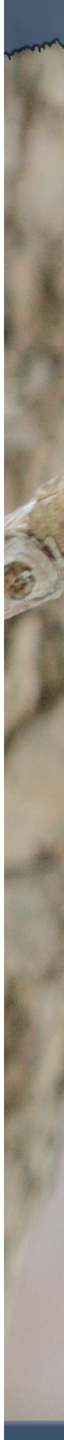
- The documentation includes provision for a *type parameter*:
  - **`ArrayList<E>`**
- These type names reappear in the parameters and return types of the methods of the class:
  - **`E get(int index)`**
  - **`boolean add(E e)`**





Parameterized classes

- The types in the documentation are placeholders for the types we use in practice:
  - An **ArrayList<Track>** actually has methods:
  - **Track get(int index)**
  - **boolean add(Track e)**



Parameterized classes

- `List l` sort of same as `List<Object> l`
- Has methods

```
Object get(int index)

boolean add(Object o)

...
```

Parameterized classes

- **`List l`** sort of same as **`List<Object> l`**
- Has methods

```
Object get(int index)

boolean add(Object o)

...
```

- **`List<String> l`**
- Has methods

```
String get(int index)

boolean add(String s)

...
```

Parameterized classes

- `List l` sort of same as `List<Object> l`
- Has methods

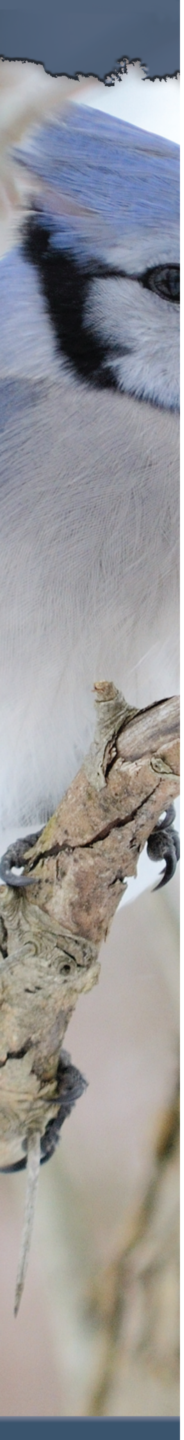
```
Object get(int index)

boolean add(Object o)

...
```

- `List<String> l`
- Has methods

```
String get(int index)

boolean add(String s)

...
```

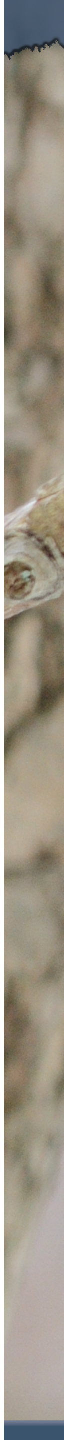
- More generally `List<E> l`
- Has methods

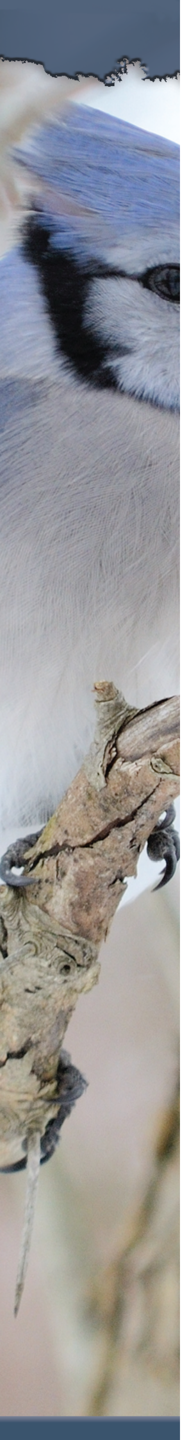
```
E get(int index)

boolean add(E e)

...
```



Review

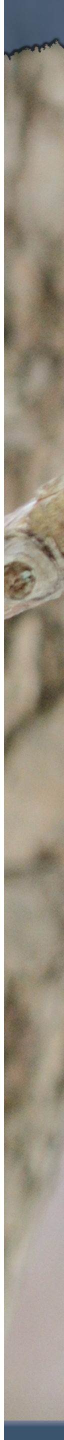
- Java has an extensive class library.
- A good programmer must be familiar with the library.
- The documentation tells us what we need to know to use a class (its interface).
- Some classes are parameterized with additional types.
  - Parameterized classes are also known as *generic classes* or *generic types*.

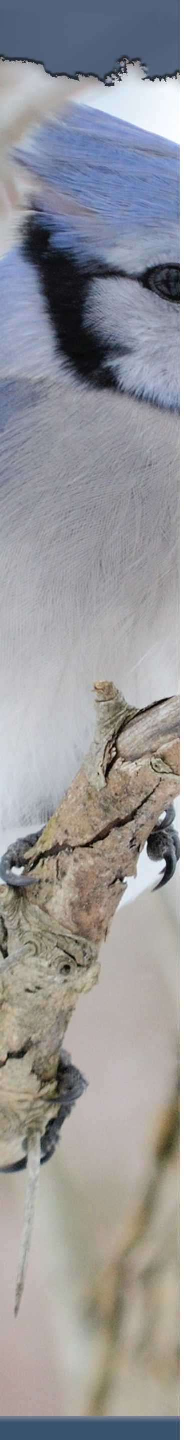




Further library classes

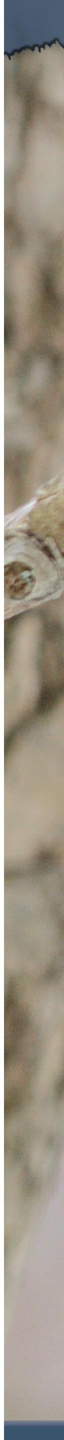
Using library classes to implement more functionality

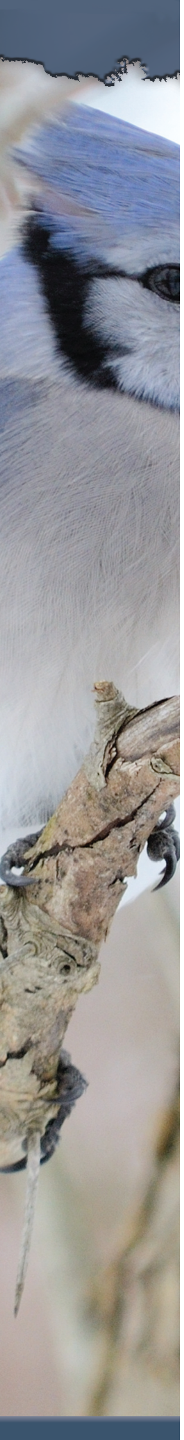




Main concepts to be covered

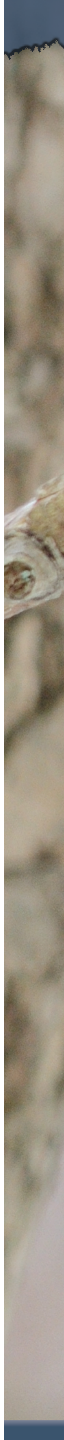
- Further library classes:
  - `Set` – avoiding duplicates
  - `Map` – creating associations
- Writing documentation:
  - `javadoc`





Writing class documentation

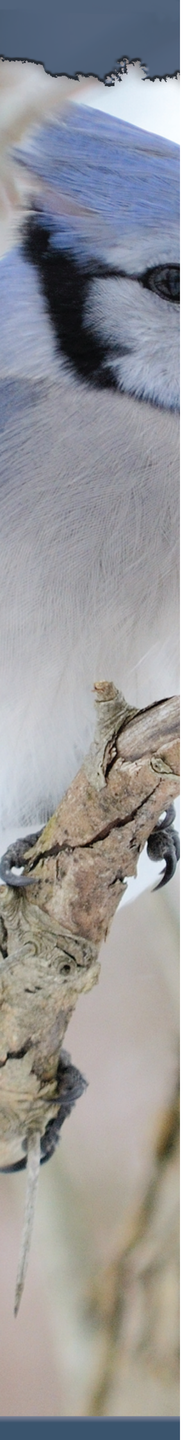
- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- Make your class a potential 'library class'!



Elements of documentation

Documentation for a class should include:

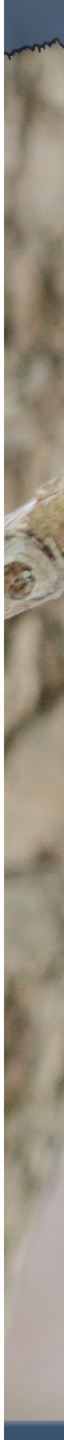
- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors' names
- documentation for each constructor and each method



Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned



Elements of documentation

But avoid documentation overkill:

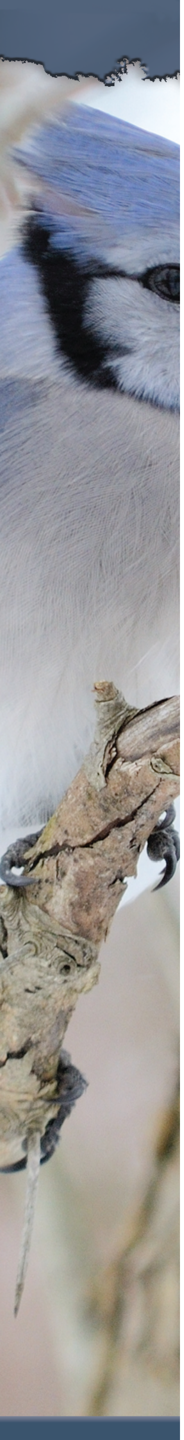
```
/**
 * Returns value of name.
 * @return value of name
 */
public String getName() {
    // returns value of name
    return name;
}
```

Code should be self-explanatory

javadoc

Class comment:

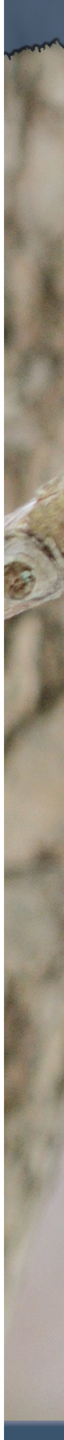
```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (2016.02.29)
 */
```

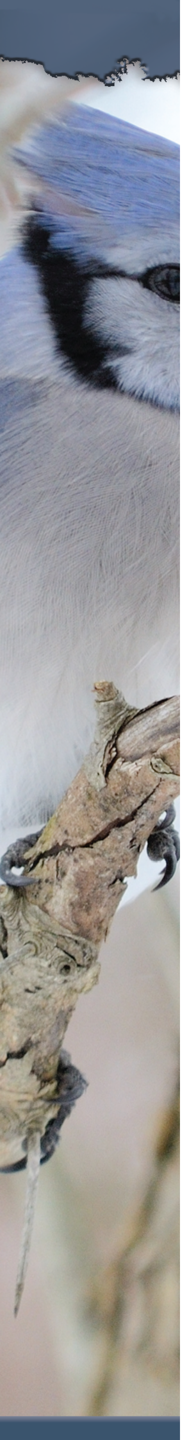


javadoc

Method comment:

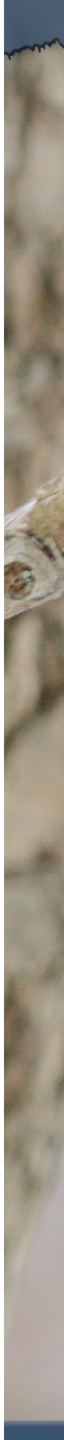
```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return A set of strings, where each String is
 *         one of the words typed by the user
 */
public HashSet<String> getInput(String prompt)  {
   ...
}
```

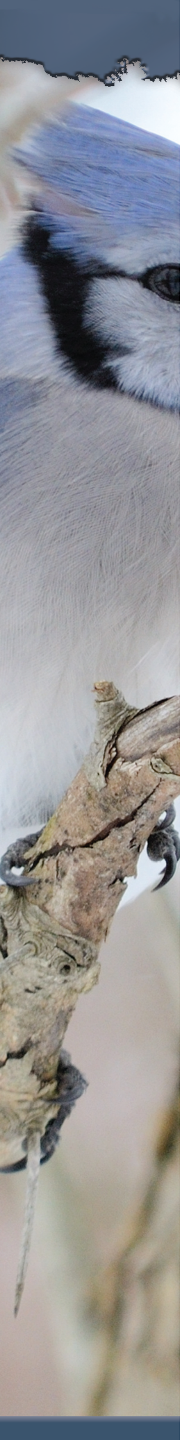




Public vs private

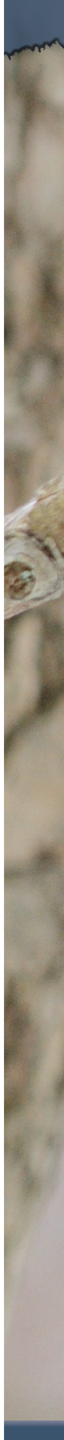
- Public elements are accessible to objects of other classes:
  - Fields, constructors and methods
- Fields should not be public.
- Private elements are accessible only to *objects of the same class*.
- Only methods that are intended for other classes should be public.





Encapsulation - data hiding

- Data belonging to one object is hidden from other objects.
- Know *what* an object can do, not *how* it does it.
- Information hiding increases the level of *independence*.
- Independence of modules is important for large systems and maintenance.



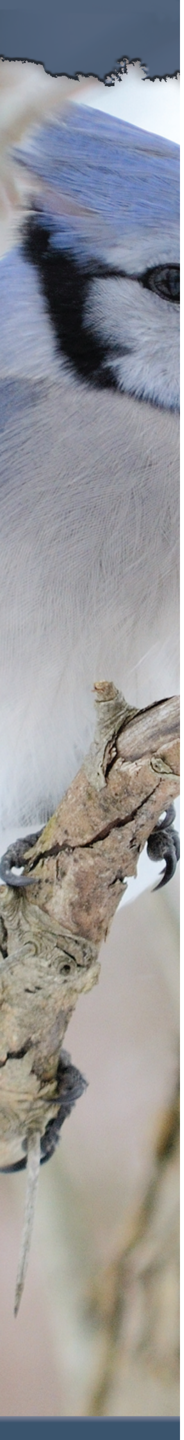
Encapsulation - method hiding

- Method access should be as restrictive as possible: private, package private, public in that order.
- Know *what* an object can do, via its public methods.
- Method hiding increases the level of *independence*.
- Independence of modules is important for large systems and maintenance.

Review

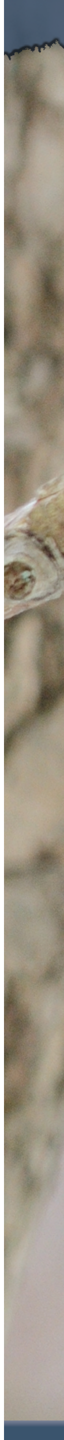
- Java has an extensive class library.
- A good programmer must be familiar with the library.
- The documentation tells us what we need to know to use a class (interface).
- The implementation is hidden (information hiding).
- We document our classes so that the interface can be read on its own (class comment, method comments).

Class variables and constants

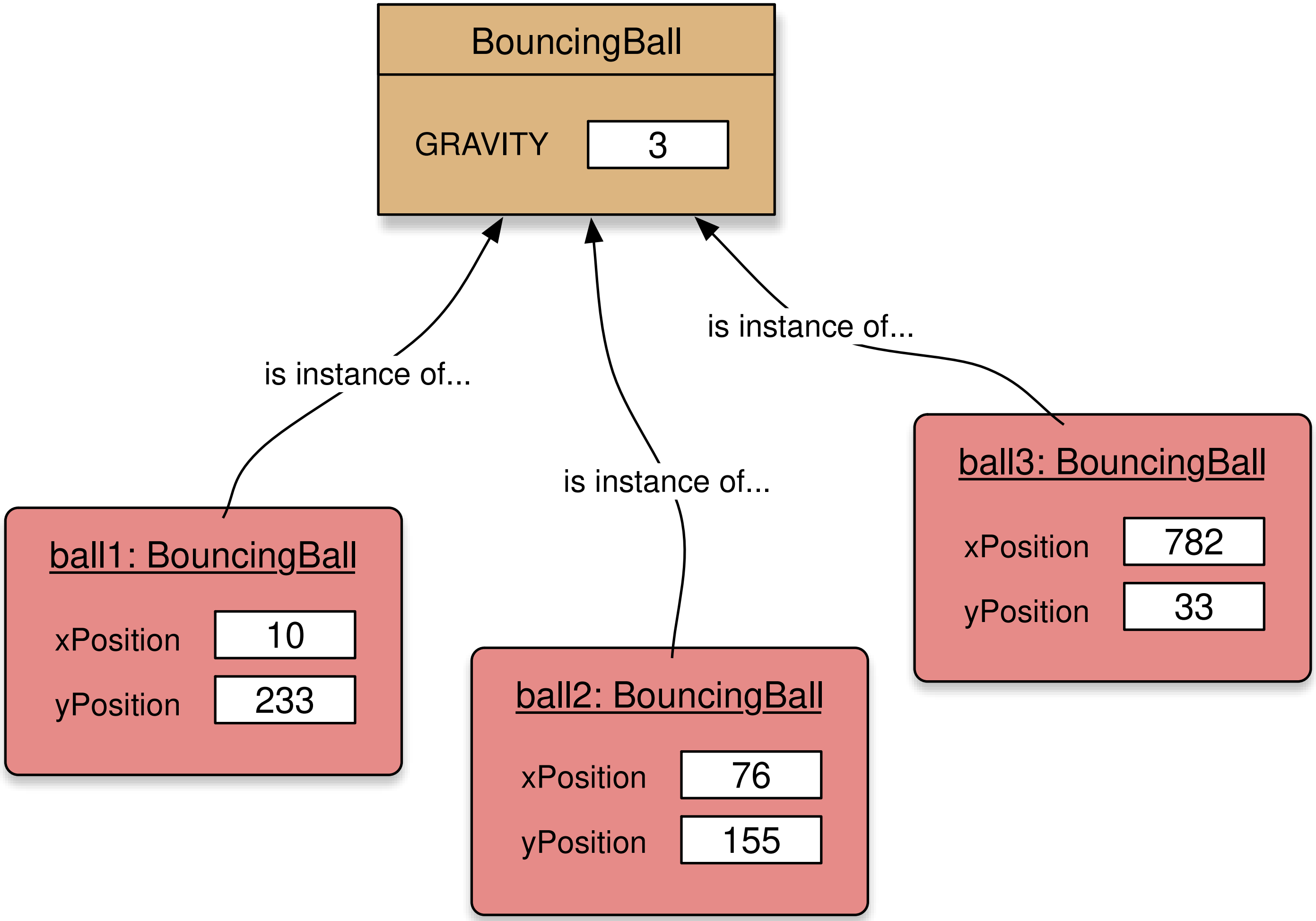


Class variables

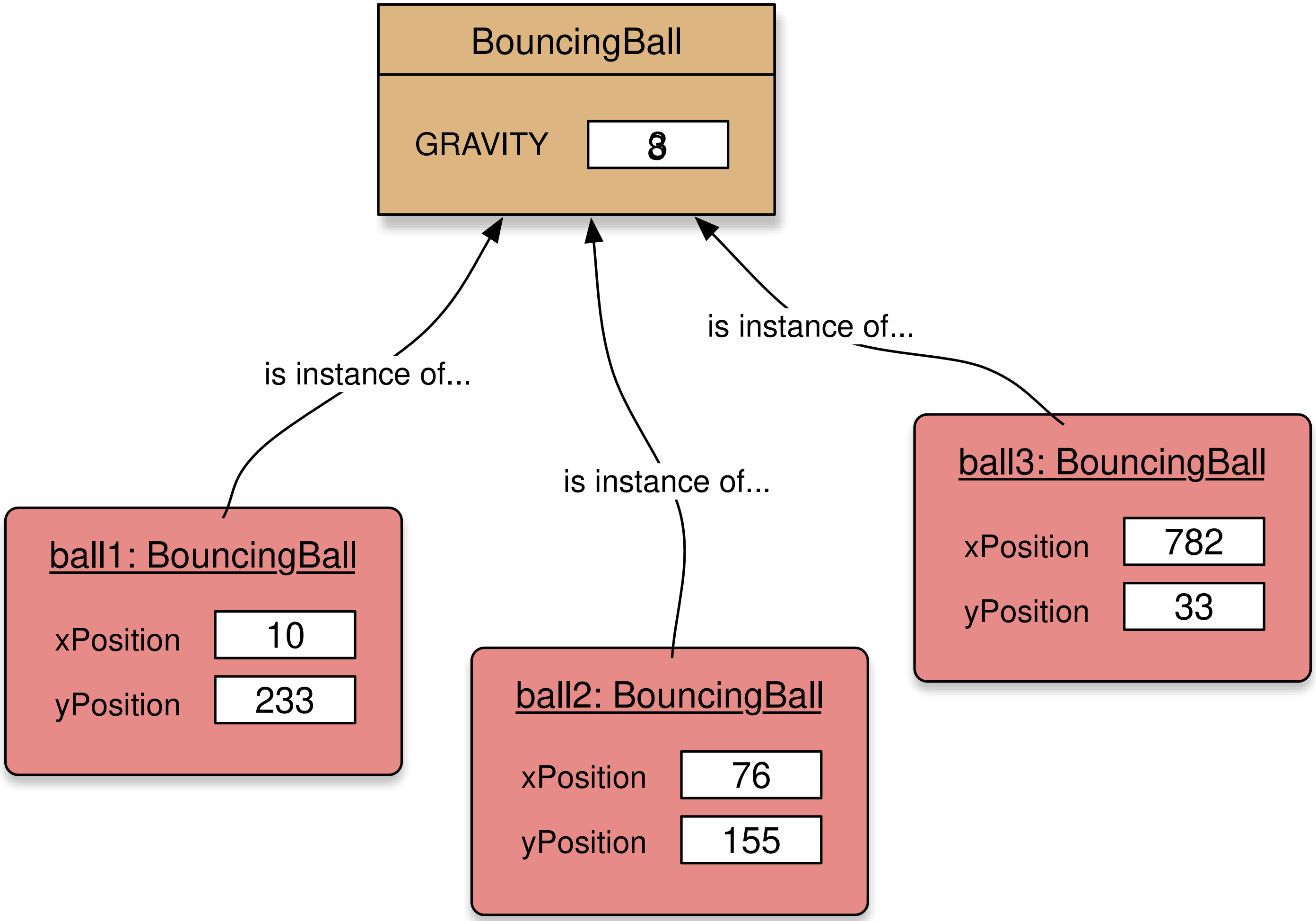
- A class variable is shared between all instances of the class.
- In fact, it belongs to the class and exists independent of any instances.
- Designated by the **`static`** keyword.
- Static variables are accessed via the class name; e.g.:
  - **`Thermometer.boilingPoint`**



Class variables



Class variables



BouncingBall

GRAVITY 8

is instance of...

is instance of...

is instance of...

ball1: BouncingBall

xPosition 10

yPosition 233

ball2: BouncingBall

xPosition 76

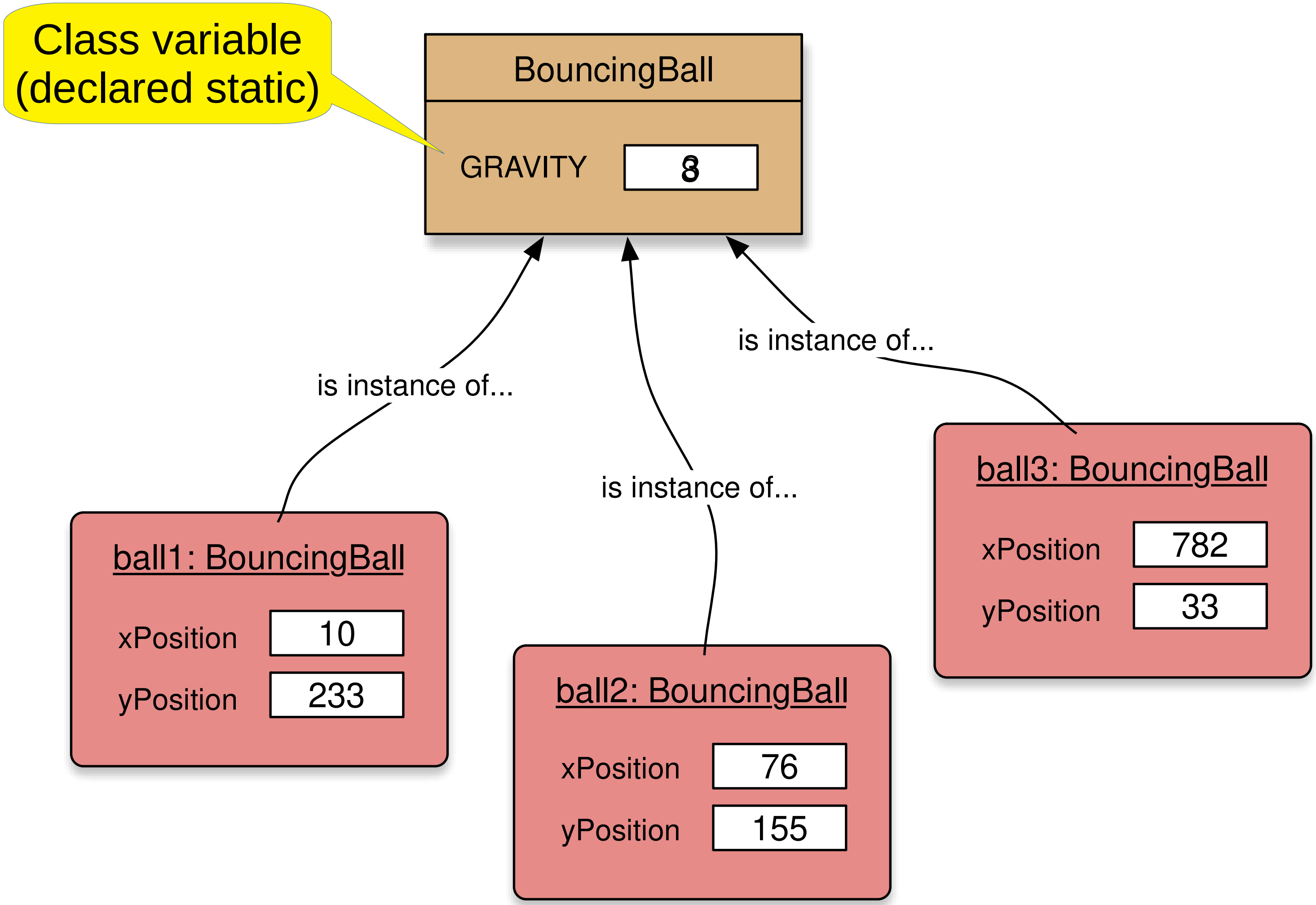
yPosition 155

ball3: BouncingBall

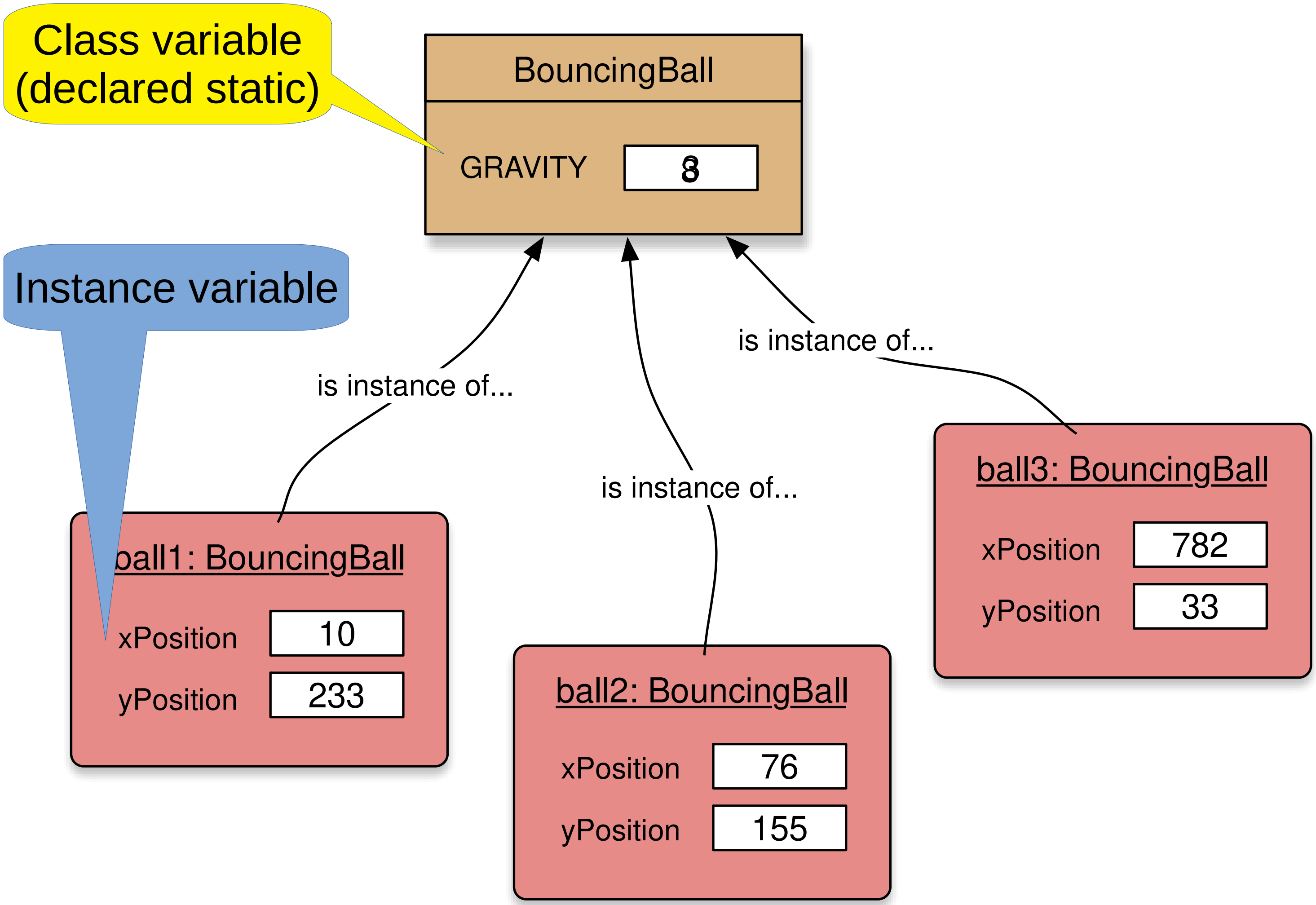
xPosition 782

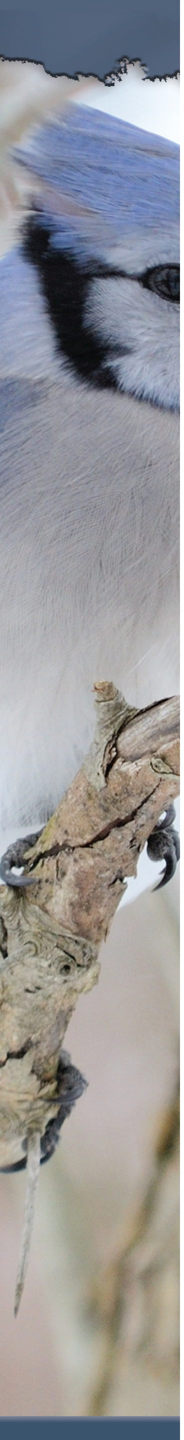
yPosition 33

Class variables



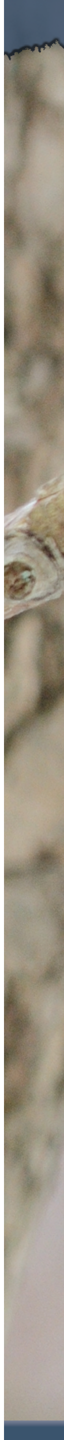
Class variables





Constants

- A variable, once set, can have its value fixed.
- Designated by the **`final`** keyword.
  - **`final int SIZE = 10;`**
- Final *fields* must be set in their declaration or the constructor.
- Combing **`static`** and **`final`** is common.



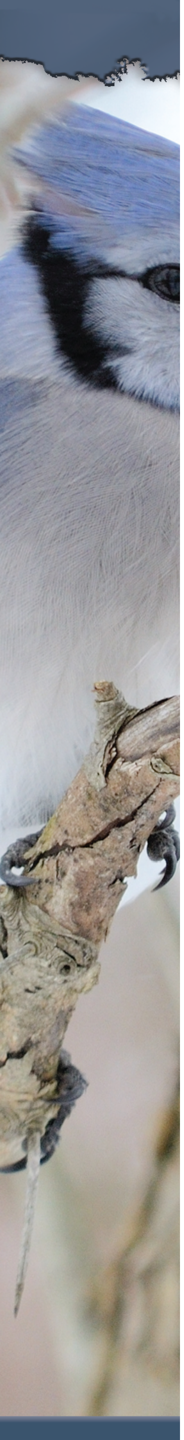
Class constants

- **`static`**: class variable
- **`static final`**: constant

```
private static final int GRAVITY = 3;
```

- Public visibility is less of an issue with **`final`** fields.
- Upper-case names often used for class constants:

```
public static final int BOILING_POINT = 100;
```



Class methods

- A **`static`** method belongs to its class rather than the instances:

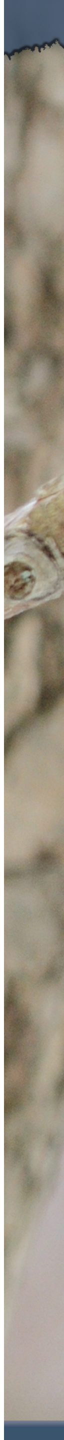
```
public static int getDaysThisMonth()
```

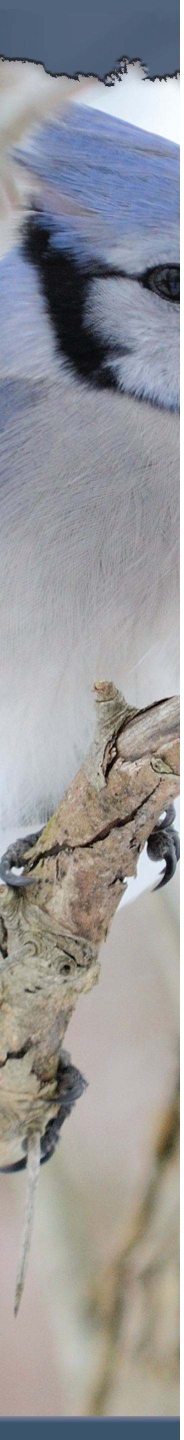
- Static methods are invoked via their class name:

```
int days = Calendar.getDaysThisMonth();
```

- ...or just by:

```
int days = getDaysThisMonth();
```





Class methods

- A **`static`** method belongs to its class rather than the instances:

```
public static int getDaysThisMonth()
```

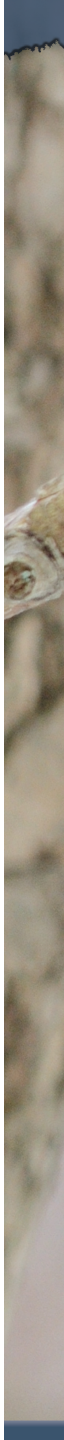
- Static methods are invoked via their class name:

```
int days = Calendar.getDaysThisMonth();
```

- ...or just by:

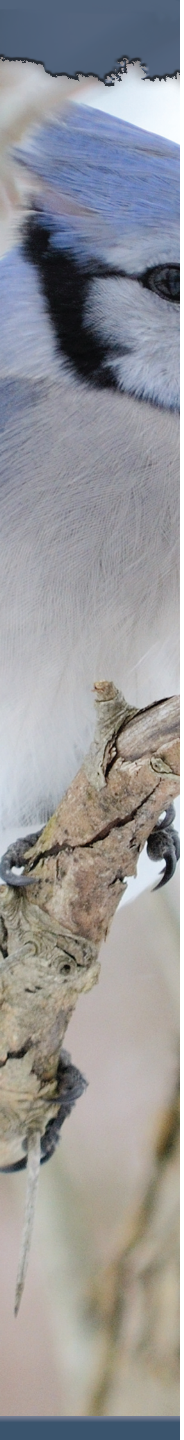
```
int days = getDaysThisMonth();
```

Not recommended – include the class name `Calendar.getDaysThisMonth`



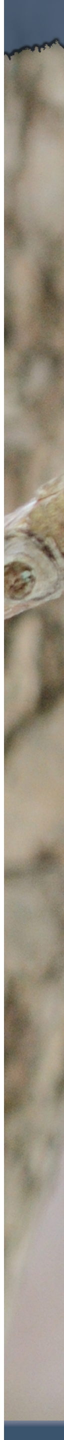
Limitations of class methods

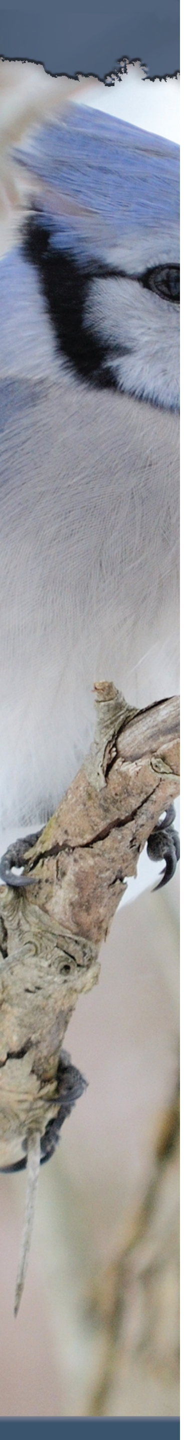
- A static method exists independent of any instances.
- Therefore:
  - They cannot access instance fields within their class.
  - They cannot call instance methods within their class.
- Should be avoided unless you have a very specific reason to use them.



Review

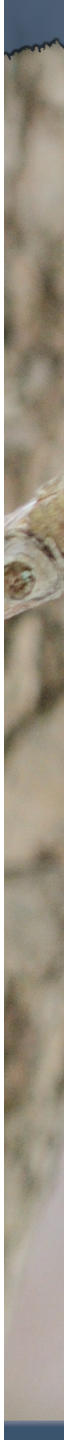
- Class variables belong to their class rather than its instances.
- Class methods belong to their class rather than its instances.
- Class variables are used to share data among instances.
- Class methods are prohibited from accessing instance variables and methods.

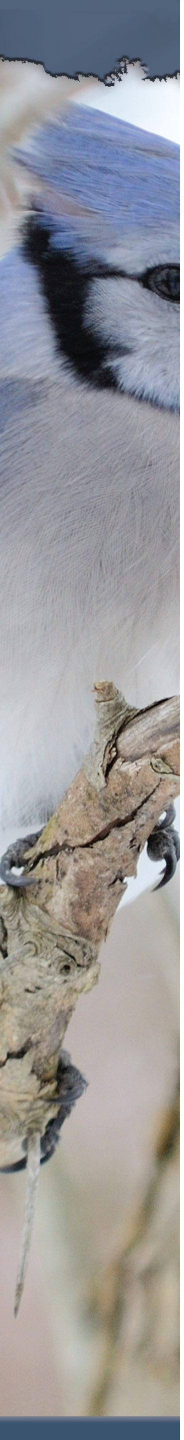


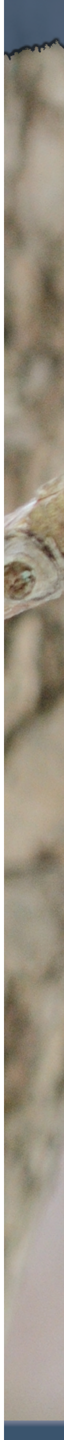


Review

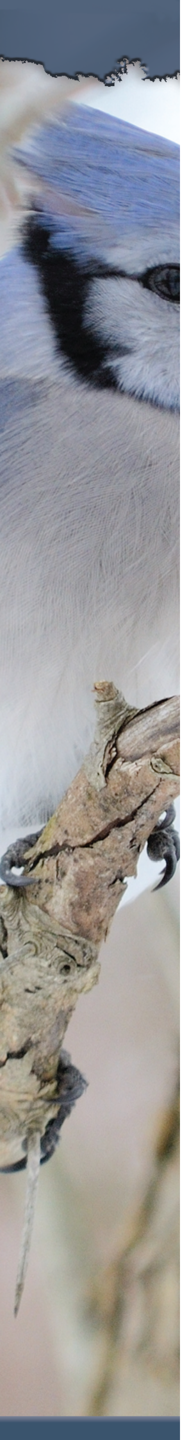
- The values of **`final`** variables are fixed.
- They must be assigned at declaration or in the constructor (for fields).
- **`final`** and **`static`** are unrelated concepts, but they are often used together to designate constants.





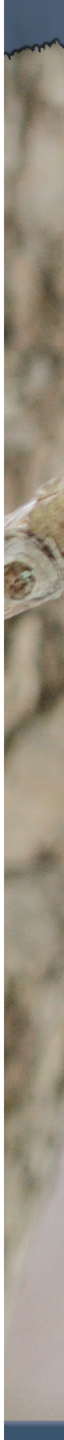


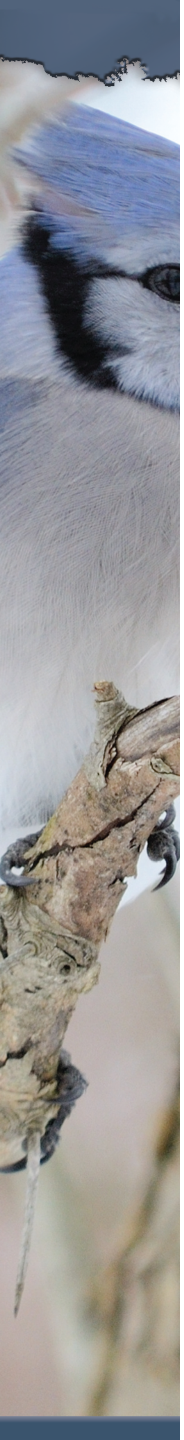
Further Advanced Material



Polymorphic collection types

- Different collection classes offer similar interfaces; e.g.:
  - **`ArrayList`** and **`LinkedList`**
  - **`HashSet`** and **`TreeSet`**
- Types exist which capture those similarities:
  - **`List`**
  - **`Set`**





Polymorphic collection types

- *Polymorphism* allows us to ignore the more specific type in most cases.
- We create objects of the specific type, but …
- … declare variables of the more general type:

```
List<Track> tracks = new LinkedList<>();
Map<String, String> responseMap =
                        new HashMap<>();
```