Class Class

Class Object

Object is the base class for all java classes. Every class *implicitly* extends Object.

Among the methods contained in Object are the following:

```
public boolean equals(Object x) {…}

public String toString( ) {…}

public Class getClass(Object x)  {…}

public int hashCode( )  {…}

protected Object clone(Object x)  {…}

public void wait( )  {…}                    //used in multithreading

public void notify( ) {…}                    //used in multithreading
```

We will examine the first five of these methods in more detail.

Class Object

Method equals( )

Class Object

Method equals( )

The default behavior of equals( ) is to compare references. You must override this method in the class you are constructing (or understand how it is overridden in the library class you are using) if you are using this method to test whether two objects have the same value.

In class String – equals( ) is overridden to test whether the two String objects have the same value.

```
String s1 = “A string”;
String s2 = “A” + new String(“ string”);
System.out.println(“Strings s1 and s2 have same reference: ”+s1==s2);
```

false

```
System.out.println(“Strings s1 and s2 have same value: ”+s1.equals(s2));
```

true

Method toString( )

Class Object

Method toString( )

The default method toString( ) contained in class Object returns a String giving

- the name of the Class and
- the address of the Object in memory.

Consider the following example

```
class ShowObject {
    private int val1, val2;
    public ShowObject(int v1, int v2) {val1 = v1; val2 = v2;}
    public static void main(String [ ] args) {
        ShowObject theObj = new ShowObject(24, 12);
        System.out.println("the output is "+theObj);
    }
}
```

the output is ShowObject@107077e

Class Object

Method toString( )

Note in the previous example that the call to

System.out.println(“some string ”+ theObj)

Results in a call to the method toString( ) in class ShowObject that is inherited from Object.

System.out.println(theObj); and

System.out.println(theObj.toString( )); are equivalent

Whenever methods print( ) or println( ) are passed an Object as an argument, they will use the object’s toString( ) method to formulate the output string.

Class Object

Method hashCode( )

Class Object

Method hashCode( )

Method hashCode( ) uses properties of an object to determine a (nearly) unique integer value.

Two objects with the same values will have different hash codes. Consider class ShowObject described on the previous slide. If we declare two objects of this class with identical values, we will obtain different hash codes.

```
pubic static void main (String [ ] args) {
    ShowObject obj1 = new ShowObject(24, 12);
    ShowObject obj2 = new ShowObject(24, 12);
    System.out.println("hash code for obj1: "+obj1.hashCode( ));        8187137
    System.out.println("hash code for obj2: "+obj2.hashCode( ));        28050664
}
```

Class Object

Method hashCode( )

A good hash function is one that uniformly distributes the keys into the range of integer values that index the hash table. For most applications, the programmer will need to override hashCode( ) derived from the base class Object.

If the programmer chooses to override method hashCode( ) in a particular class, he or she must supply

1. **Supply a (private) hash function that maps attributes of the object into integer values.**
2. **Choose some attribute or attributes of the class that form the domain of the hash function.**

Class Object

Method clone( )

Class Object

Method clone( )

Not all classes can be cloned! clone( ) is a protected method in Base class Object, and hence is not directly accessible to a client application.

Protected methods are accessible to derived classes and hence may be overridden and made public.

- **An application can create a clone of some object only if it has access to the clone( ) method in the class to which that object belongs.**
- **Unless method clone( ) is overriden, a derived class will inherit clone( ) from Object, but the method is protected and not accessible to a client – therefore no clones of objects of that class can be created.**
- **To allow clients to create clones of objects of a particular class, method clone( ) will have to be overridden and made public.**
- **Once clone( ) is made public in a class, it is public also for any classes derived from that class.**

Class Object

Method clone( )

There are three important rules to follow if you are going to override clone( ) in a class you are creating.

1. (Virtually) Always call super.clone( ) -- the base class clone( ) method performs the bitwise duplication of the derived class object. (If the attributes are all primitive types it is safe to not call super.)

2. Make your clone( ) method public

3. Implement the Cloneable interface

   •To determine whether an instance of an object can be cloned

   if (myReference instanceof Cloneable) {..}

   •Cloneable is an "empty interface" with no methods to implement. It acts as a "flag" identifying that an object can be cloned.

Class Object

Method clone( )

```
class Object

protected Object clone( ) {
   //perform bitwise copy
}

//other methods
```

```
class MyThing extends Object
implements Cloneable
```

```
//data attributes
private int  myInt;  //primitive type
private YourThing  aThing; //object
```

```
public Object clone ( ) {
    MyThing result = null;
    result = (MyThing)super.clone( );
    result.aThing = (YourThing)
            result.aThing.clone( );
    return result;
}

//other methods
```

Here we assume the object attribute is Cloneable, and we convert the bitwise copy done in the call to super – that only copies a handle to an object – into a "deep copy" of the object itself by calling the object's own clone( ) method.

Class Object

Method clone( )

Some frequently used Java library classes are not cloneable.

- **The wrapper classes (Integer, Double, etc.) do not override method clone( ) and hence cannot be cloned.**

- **String and StringBuffer classes are not Cloneable.**

- **Standard Containers, such as *arrayList*, can be cloned, but a shallow copy of their contents is made because it cannot be certain that those contained objects can be cloned.**

Class Object

Method clone( ) – an example

A Box is an unordered collection of objects of fixed size.

```
public class Box implements Cloneable {
    protected Object [ ] box;
    protected int size;
    public Box(int bsize) {
        box = new Object[bsize];
        size = 0;
    }
    public Object clone ( ) {
        Box  result = null;
        try {
            result = (Box)super.clone( );
        } catch (CloneNotSupportedException e) {System.out.println(e); }
        for (int i = 0; i < box.length; i++)  //override bitwise copy if possible
            if (box[i]  instanceof Cloneable)       { result.box[i] = box[i].clone( ); }
        return result;
    }
}
```

You need to enclose the call to super.clone inside a try block

Class Object

Method getClass( )

Class Object

Method getClass( )

This method is useful when you have a container holding objects of a base class (such as Shape) some or all of which are instances of a derived class (such as Circle or Rectangle). The method getClass( ) allows one to determine which kind of an object each is, and determine whether a particular down cast is appropriate.

This method is inherited from the base class Object and should never be overridden in any derived class.

Class Object

Method getClass( ) -- Example

Let Shape be an abstract class and Circle and Rectangle concrete classes that extend Shape.

```
public static void main (String [ ] args) {
    Shape [ ] shapeList = new Shape [3];
    shapeList[0] = new Circle(5.0);
    shapeList[1] = new Rectangle(3.0, 4.0);
    shapeList[2] = new Circle(6.0);
    for (int i = 0; i < shapeList.length; i++) {
        Class c = shapeList[i].getClass( );
        if (c.equals(Circle.class))
            System.out.println("I can be cast as a Circle");
        else if (c.equals(Rectangle.class))
            System.out.println("I can be cast as a Rectangle");
    }
}
```

Find out to which class each object in shapeList belongs

Knowing the class that an object belongs to allows the client programmer to down cast the object and send messages to methods particular to that derived class.

RTTI

The idea of run-time type identification (RTTI) seems fairly simple at first: it lets you find the exact type of an object when you have a handle to only the base type.

Class object

- The Class object is used to create all of the "regular" objects of your class. Each time you write a new class, a single Class object is also created (and stored, appropriately enough, in an identically named .class file). At run time, when you want to make an object of that class, the Java Virtual Machine (JVM) that's executing your program first checks to see if the Class object for that type is loaded. If not, the JVM loads it by finding the .class file with that name.
- Case Study: SweetShop.java

- The output of this program for one JVM is:
  - inside main Loading Candy
  - After creating Candy
  - Loading Gum
  - After Class.forName("Gum")
  - Loading Cookie
  - After creating Cookie

The Class Object

- Class literals also provide a reference to the Class object
  - *E.g.* `Gum.class`
- Each object of a primitive wrapper class has a standard field called TYPE that also provides a reference to the Class object
  - http://java.sun.com/j2se/1.3/docs/api/java/lang/Boolean.html

Instance of

- Java 1.1 has added the isInstance method to the class Class.
- Case Study: PetCount3.java

Class Object

Method getClass( )

Note that there are alternative ways of determining the class of an object. To fully understand how these determinations are made, you will need to become familiar with the Java RTTI (Run-time type identification) which requires understanding of the subtle distinctions between the base *class* Object from which all classes are derived and Class objects that are created when a class is compiled into bytecode. Class objects are stored in the .class file.

Let x be an object of one of the derived classes in the previous example. Then we may determine the class of this object by:

```
Class c = x.getClass( );    if (c.equals(Circle.class)){ }
```

← Compares 2 classes

Tests whether references are the same -- c == Circle.class is equivalent

```
if (x instanceof Circle) { }
```

← Tests whether object x is an instance of a class

instanceof is a comparator (an operator)

```
if (c.isInstance(x) ) { }
```

← Alternatively (but equivalently) use the isInstance( ) method of an object with a class parameter

RTTI mothods

- **Case Study: ToyTest.java**

Reflection: run-time class information

- Where to use Reflection?
  - Component-based programming in which you build projects using Rapid Application Development (RAD) in an application builder tool.
    - To provide the ability to create and execute objects on remote platforms across a network. This is called Remote Method Invocation (RMI) and it allows a Java program (version 1.1 and higher) to have objects distributed across many machines.
- Case Study: ShowMethods.java

Get Filed Value

To print the names of the ViolinNote fields as well as their current values in the particular ViolinNote object referenced by note:

```
Field fields[] = c.getFields(); try {
 for(int i = 0; i < fields.length; i++) {
  System.out.print(fields[i].getName() + " =
  ");
  System.out.println(fields[i].getInt(note))
  ;
 }
} catch(Exception e) {
}
```

Invoke Method

```
note = new Note();
c = note.getClass();
Method meth = c.getMethod("play",
null);
meth.invoke(note, null);
```

Object Serialization

- allows you to take any object that implements the Serializable interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object.

- To serialize an object, you create some sort of OutputStream object and then wrap it inside an ObjectOutputStream object. At this point you need only call writeObject( ), and your object is serialized and sent to the OutputStream. To reverse the process, you wrap an InputStream inside an ObjectInputStream and call readObject( ). What comes back is, as usual, a reference to an upcast Object, so you must downcast to set things straight.

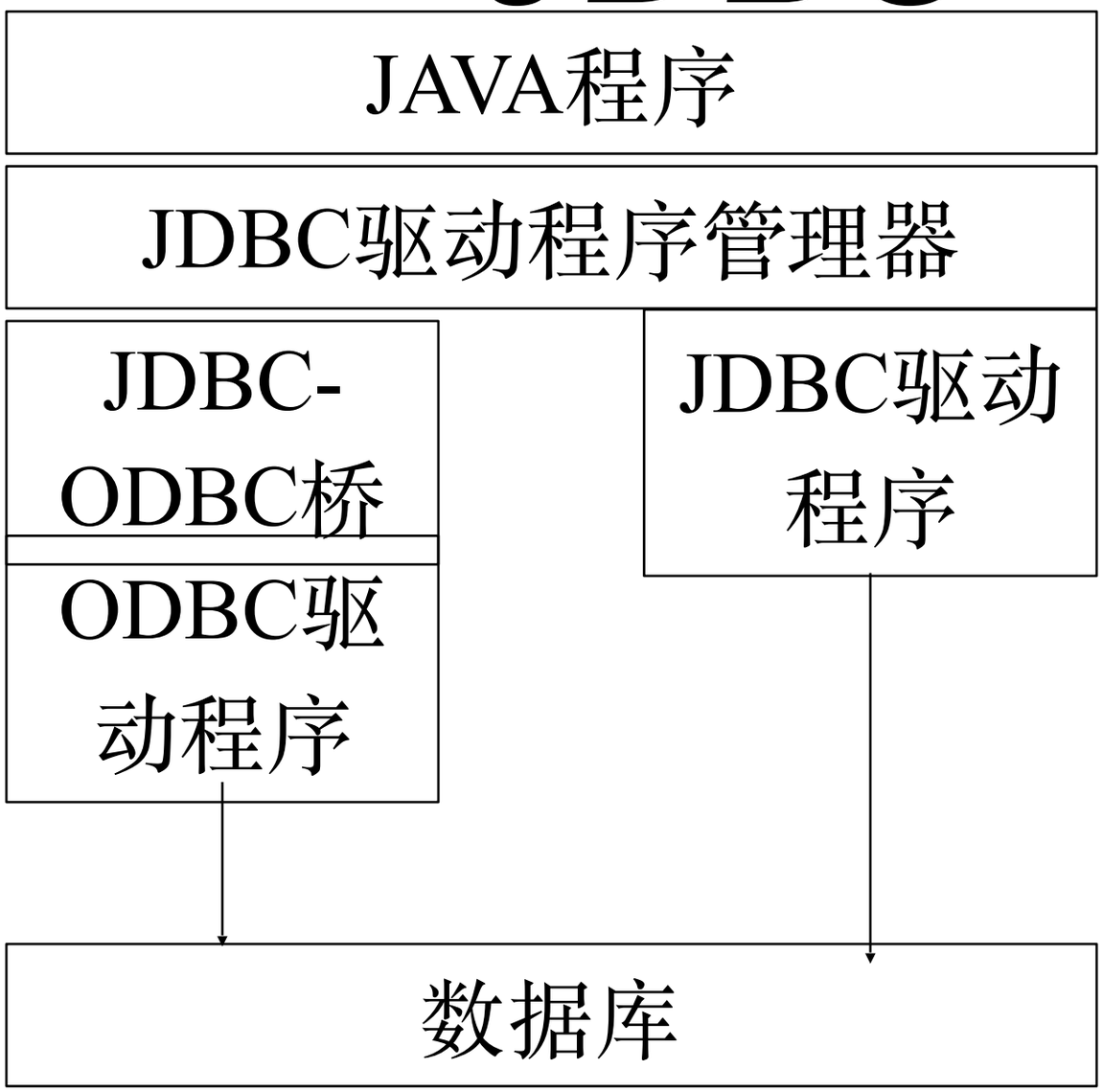
- Case: Worm.java

串行化和RTTI

- 反串行化回来的对象为什么还是能用的?

JDBC

JDBC



- Java DataBase Connection
- 遵循ODBC模型，但不是ODBC

JDBC三步曲

* DriverManager
* Connection
* Statement
* ResultSet
* Case Study:Lookup.java

更新数据库

- Statement.executeUpdate(String SQL);

PreparedStatement

- 由Connection产生PreparedStatement对象，其中的参数用?表示
- 用PreparedStatement的setXXX()函数对参数赋值
- 用PreparedStatement的execute()执行
- Case: Query DB.java

事务处理

- 在Connection上做事务处理
  - setAutoCommit()
  - commit()
  - rollback()

JavaDoc

JavaDoc

- The JavadocTM tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing (by default) the public and protected classes, nested classes (but not anonymous inner classes), interfaces, constructors, methods, and fields. You can use it to generate the API (Application Programming Interface) documentation or the implementation documentation for a set of source files.

DOCUMENTATION COMMENTS

- /**
  * ……..
  */
  public int getCount() { .....

- **Sample: Image.java**

Placement of comments

- Documentation comments are recognized only when placed immediately before class, interface, constructor, method, or field declarations. Documentation comments placed in the body of a method are ignored. Only one documentation comment per declaration statement is recognized by the Javadoc tool.

- A doc comment is composed of a main description followed by a tag section - The main description begins after the starting delimiter /** and continues until the tag section. The tag section starts with the first block tag, which is defined by the first @ character that begins a line (ignoring leading asterisks, white space, and leading separator /**). It is possible to have a comment with only a tag section and no main description. The main description cannot continue after the tag section begins. The argument to a tag can span multiple lines. There can be any number of tags.

- A tag is a special keyword within a doc comment that the Javadoc tool can process. There are two kinds of tags: block tags, which appear as @tag (also known as "standalone tags"), and in-line tags, which appear within curly braces, as {@tag}. To be interpreted, a block tag must appear at the beginning of a line. This means you can use the @ character elsewhere in the text and it will not be interpreted as the start of a tag. If you want to start a line with the @ character and not have it be interpreted, use the HTML entity @. Each block tag has associated text, which includes any text following the tag up to, but not including, either the next tag, or the end of the doc comment.

- Each line above is indented to align with the code below the comment.
- The first line contains the begin-comment delimiter (/**).
- Starting with Javadoc 1.4, the leading asterisks are optional.
- Write the first sentence as a short summary of the method, as Javadoc automatically places it in the method summary table (and index).
- Notice the inline tag {@link URL}, which converts to an HTML hyperlink pointing to the documentation for the URL class. This inline tag can be used anywhere that a comment can be written, such as in the text following block tags.
- If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag, as shown.
- Insert a blank comment line between the description and the list of tags, as shown.
- The first line that begins with an "@" character ends the description. There is only one description block per doc comment; you cannot continue the description following block tags.
- The last line contains the end-comment delimiter (*/) Note that unlike the begin-comment delimiter, the end-comment contains only a single asterisk.

Leading asterisks

- When javadoc parses a doc comment, leading asterisk (*) characters on each line are discarded; blanks and tabs preceding the initial asterisk (*) characters are also discarded. Starting with 1.4, if you omit the leading asterisk on a line, the leading white space is no longer removed.

First sentence

- The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the declared entity. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first block tag. The Javadoc tool copies this first sentence to the member summary at the top of the HTML page.

Declaration with multiple fields

- Java allows declaring multiple fields in a single statement, but this statement can have only one documentation comment, which is copied for all fields. Therefore if you want individual documentation comments for each field, you must declare each field in a separate statement.

Use header tags carefully

- When writing documentation comments for members, it's best not to use HTML heading tags such as <H1> and <H2>, because the Javadoc tool creates an entire structured document and these structural tags might interfere with the formatting of the generated document.

JAVADOC TAGS

- The Javadoc tool parses special tags when they are embedded within a Java doc comment. These doc tags enable you to autogenerate a complete, well-formatted API from your source code. The tags start with an "at" sign (@) and are case-sensitive -- they must be typed with the uppercase and lowercase letters as shown.

- Block tags - Can be placed only in the tag section that follows the main description. Block tags are of the form: @tag.
- Inline tags - Can be placed anywhere in the main description or in the comments for block tags. Inline tags are denoted by curly braces: {@tag}.

- @author name-text
- @deprecated deprecated-text
- @exception class-name description
- @param parameter-name description
- @return description

- @see reference
  - @see "string"
    - @see "The Java Programming Language"
  - @see label
    - @see Java Spec
  - @see package.class#member label
    - @see String#equals(Object) equals

- @since since-text
- @throws class-name description
- @version version-text

Where Tags Can Be Used

- these tags can be used in all doc comments: @see, @since, @deprecated, {@link}, {@linkplain}, and {@docroot}.

Overview Documentation Tags

- @see
- @since
- @author
- @version
- {@link}
- {@linkplain}
- {@docRoot}

Package Documentation Tags

- @see
- @since
- @serial
- @author
- @version
- {@link}
- {@linkplain}
- {@docRoot}

Class and Interface Documentation Tags

- @see
- @since
- @deprecated
- @serial
- @author
- @version
- {@link}
- {@linkplain}
- {@docRoot}

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *     Window win = new Window(parent);
 *     win.show();
 * </pre>
 *
 * @author  Sami Shaio
 * @version %I%, %G%
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
   ...
```

Field Documentation Tags

- @see
- @since
- @deprecated
- @serial
- @serialField
- {@link}
- {@linkplain}
- {@docRoot}
- {@value}

- @see Constructor and Method
- @since Documentation Tags
- @deprecated
- @param
- @return
- @throws and @exception
- @serialData
- {@link}
- {@linkplain}
- {@inheritDoc}
- {@docRoot}

```
/**
    * Returns the character at the specified index. An index
    * ranges from <code>0</code> to <code>length() - 1</code>.
    *
    * @param     index  the index of the desired character.
    * @return    the desired character.
    * @exception StringIndexOutOfRangeException
    *                if the index is not in the range <code>0</code>
    *                to <code>length()-1</code>.
    * @see       java.lang.Character#charValue()
    */
public char charAt(int index) {
   ...
}
```

Order of Block Tags

1. @author
2. @version
3. @param
4. @return
5. @exception
6. @see ,
7. @since
8. @serial
9. @deprcated

A Style Guide

- **Use <code> style for keywords and names.** Keywords and names are offset by <code>...</code> when mentioned in a description. This includes:
  - Java keywords
  - package names
  - class names
  - method names
  - interface names
  - field names
  - argument names
  - code examples

- **Use in-line links economically**
  You are encouraged to add links for API names (listed immediately above) using the {@link} tag. It is not necessary to add links for *all* API names in a doc comment. Because links call attention to themselves (by their color and underline in HTML, and by their length in source code doc comments), it can make the comments more difficult to read if used profusely. We therefore recommend adding a link to an API name if:
  - The user might actually want to click on it for more information (in your judgment), and
  - Only for the first occurrence of each API name in the doc comment (don't bother repeating a link)
- Our audience is advanced (not novice) programmers, so it is generally not necessary to link to API in the java.lang package (such as String), or other API you feel would be well-known.

- **Omit parentheses for the general form of methods and constructors**
  When referring to a method or constructor that has multiple forms, and you mean to refer to a specific form, use parentheses and argument types. For example, ArrayList has two add methods: add(Object) and add(int, Object).

  The add(int, Object) method adds an item at a specified position in this arraylist.

However, if referring to both forms of the method, omit the parentheses altogether. It is misleading to include empty parentheses, because that would imply a particular form of the method. The intent here is to distinguish the general method from any of its particular forms. Include the word "method" to distinguish it as a method and not a field.

The add method enables you to insert items. (preferred)

The add() method enables you to insert items. (avoid when you mean "all forms" of the add method)

- **Okay to use phrases instead of complete sentences, in the interests of brevity.** This holds especially in the initial summary and in @param tag descriptions.

- **Use 3rd person (descriptive) not 2nd person (prescriptive).**
  The description is in 3rd person declarative rather than 2nd person imperative.
  - Gets the label. (preferred)
  - Get the label. (avoid)

- **Method descriptions begin with a verb phrase.**
  A method implements an operation, so it usually starts with a verb phrase:
  - Gets the label of this button. (preferred)
  - This method gets the label of this button. (avoid)

- **Class/interface/field descriptions can omit the subject and simply state the object.** These API often describe things rather than actions or behaviors:
  - A button label. (preferred)
  - This field is a button label. (avoid)

- **Use "this" instead of "the" when referring to an object created from the current class.** For example, the description of the getToolkit method should read as follows:
  - Gets the toolkit for this component. (preferred)
  - Gets the toolkit for the component. (avoid)

- **Add description beyond the API name.** The best API names are "self-documenting", meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information. For example, if method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name.

- Avoid

```
/**
* Sets the tool tip text.
*
* @param text the text of the tool tip
*/
public void setToolTipText(String text) {
```

- Preferred

```
/**
* Registers the text to display in a tool tip. The text
* displays when the cursor lingers over the component.
*
* @param text the string to display. If the text is null,
* the tool tip is turned off for this component.
*/
public void setToolTipText(String text) {
```

- **Be clear when using the term "field".** Be aware that the word "field" has two meanings:
  - static field, which is another term for "class variable"
  - text field, as in the TextField class. Note that this kind of field might be restricted to holding dates, numbers or any text. Alternate names might be "date field" or "number field", as appropriate.

- **Avoid Latin** -- use "also known as" instead of "aka", use "that is" or "to be specific" instead of "i.e.", use "for example" instead of "e.g.", and use "in other words" or "namely" instead of "viz."

Annotations

- *Annotations* provide data about a program that is not part of the program, such as naming the author of a piece of code or instructing the compiler to suppress specific errors. An annotation has no effect on how the code performs.

- Annotations use the form *@annotation* and may be applied to a program's declarations: its classes, fields, methods, and so on. The annotation appears first and often (by convention) on its own line, and may include optional arguments:

```
@Author("MyName")
class myClass() { }
```

or

```
@SuppressWarnings("unchecked")
void MyMethod() { }
```

- there are three built-in annotations that every Java programmer should know:
  - @Deprecated,
  - @Override,
  - @SuppressWarnings.

```
import java.util.List;
class Food {}
class Hay extends Food {}
class Animal {
    Food getPreferredFood() { return null; }
    /**
    *@deprecated document why the method was deprecated
    */
    @Deprecated
    static void deprecatedMethod() { }
}
class Horse extends Animal {
    Horse() { return; }
    @Override
    Hay getPreferredFood() { return new Hay(); }
    @SuppressWarnings("deprecation")
    void useDeprecatedMethod() {
        Animal.deprecateMethod(); //deprecation warning - suppressed
    }
}
```

@Deprecated

- The @Deprecated annotation indicates that the marked method should no longer be used. The compiler generates a warning whenever a program uses a deprecated method, class, or variable. When an element is deprecated, it should be documented using the corresponding @deprecated tag, as shown in the preceding example. Notice that the tag starts with a lowercase "d" and the annotation starts with an uppercase "D". In general, you should avoid using deprecated methods — consult the documentation to see what to use instead.

@Override

- The @Override annotation informs the compiler that the element is meant to override an element declared in a superclass. In the preceding example, the override annotation is used to indicate that the getPreferredFood method in the Horse class overrides the same method in the Animal class. If a method marked with @Override fails to override a method in one of its superclasses, the compiler generates an error.

- While it's not required to use this annotation when overriding a method, it can be useful to call the fact out explicitly, especially when the method returns a subtype of the return type of the overridden method. This practice, called *covariant return types*, is used in the previous example: Animal.getPreferredFood returns a Food instance. Horse.getPreferredFood (Horse is a subclass of Animal) returns an instance of Hay (a subclass of Food).

@SuppressWarnings

- The @SuppressWarnings annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the previous example, the useDeprecatedMethod calls a deprecated method of Animal. Normally, the compiler generates a warning but, in this case, it is suppressed.

- Every compiler warning belongs to a category. The Java Language Specification lists two categories: "deprecation" and "unchecked". The "unchecked" warning can occur when interfacing with legacy code written before the advent of generics. To suppress more than one category of warnings, use the following syntax:
- @SuppressWarnings({"unchecked", "deprecation"}) Consult your compiler's documentation for a complete list of supported warning categories.