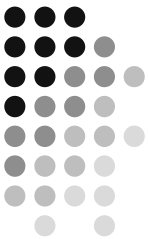
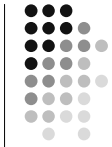
Lecture 10

Documentation,
Garbage Collection,
and Nested Classes/Interfaces

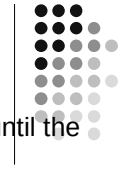


Documentation Comments Overview



- The Java standard APIs are shown in HTML output at http://java.sun.com/j2se/1.4.2/docs/api/index.html. It's generated from the *documentation comments (doc comments)*.
- Documentation comments are special comments in the source code that are delimited by the /** ... */ delimiters.
- The JDK contains a tool named `javadoc` to generate HTML documentation from documentation comments in your source file. The `javadoc` utility extracts information for the following items
  - Public classes and interfaces
  - Public and protected methods
  - Public and protected fields
  - Packages

Details on the Doc Comments



- Doc comments star with the three characters /** and continue until the next */.
  - E.g

```
/**
 * Do what the invoker intends. “Intention” is defined by
 * an analysis of past behavior as described in ISO 4074-6
 */
public void dwim() throws IntentUnknownException;
```

  - Leading * characters, and their preceding white spaces are ignored
  - The first sentence of the comment is the summary for the identifier.
  - You can use most of the HTML tags in the text formatting or providing cross-reference links to other documentation.
  - Only doc comments that IMMEDIATELY PRECEDE a class/interface, method, or field are processed.
  - If no doc comment is given for an inherited method, the method inherits the doc comments from its supertype
  - If a method inherits doc comments from both a superclass and superinterface, the interface comment are used.

Tags in the Doc Comments

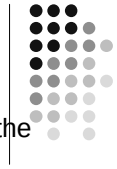
- `@see`: creates a cross-reference link to other javadoc documentation. It’s used in a “See also” section at the end of the documentation.
  - Qualify the identifier **sufficiently**.
    - specify class/interface members by using a # before the member name. If a method is overloaded, list its parameters.
    - Specify classes/interfaces with their simple names. If a class/interface is from another package, specify its package name.
    - Examples:

```
@see #getName
@see Attr
@see com.hostname.attr.Attr
@see com.hostname.attr.Attr#getName
@see com.hostname.attr.Attr#Attr(String, Object)
@see com.hostname.attr.Attr#Attr(String)
@see <a href=“spec.html#attr”>Attribute Specification</a>
```

    - You can also use a *label* after an entity reference. The label will be the actual text displayed.

```
@see #getName Attribute Names
```

Tags in the Doc Comments (cont.)



- `{@link}`: similar to `@see`, but it embeds a cross reference in the text of your comments
  - Syntax: `{@link package.class#member [label]}`
  - The identifier specification follows the same requirement for `@see`
  - Example:

```
Changes the value returned by calls to {@link #getValue}
```

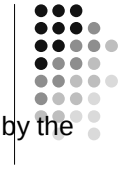
- `{@param}`: documents a single parameter of a method
  - If you use `@param` tags, you should have one for each parameter of the method
  - Syntax: `@param parameter-name description`
  - Example:

```
@param max The maximum number of words to be read
```

- `{@return}`: documents the return value of a method
  - Example:

```
@return The number of words actually read
```

Tags in the Doc Comments (cont.)



- `@throws` and `@exception`: documents an exception thrown by the method.
  - If you use `@throws` tags, you should have one for each type of exception the method throws.
  - Example:

```
@throws NullPointerException The name is <code>null</code>
```

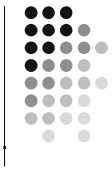
- `@deprecated`: marks that an identifier should no longer be used. It should suggest a replacement.
  - Example:

```
@deprecated Use <code>setVisible(true)</code>instead
```

- `@author`
  - Only one author name per `@author` paragraph
- `@version`
- `@since`: denote when the tagged entity was added to your system
- Example: Graphics.java Output Documentation

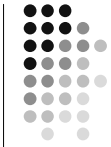
```
> javadoc Graphics.java
```

Package Documentation



- Unlike doc comments, packages are not defined in source files.
- To generate package comments, you need to add a `package.html` file in the package directory.
  - The contents of the `package.html` between `<body>` and `</body>` will be read as if it were a doc comment.
  - `@deprecated`, `@author`, and `@version` are not used in a package comment
  - The first sentence of the body is the summary of the package.
  - Any `@see` and `{@link}` tag must use the fully qualified form of the entity's name, even for classes and interfaces within the package itself.
- You can also provide an overview comment for all source files by placing a `overview.html` file in the parent directory
  - The contents between `<body>` and `</body>` is extracted
  - The comment is displayed when the user selects "`Overview`"

Garbage Collection



- Objects are created using `new`, but there is no corresponding `delete` operation to reclaim the memory used by an object.
- The Java virtual machine used garbage collection to ensure that any referenced object will remain in memory, and to free up memory by deallocating objects that are no longer reachable from references in executing code.
- Garbage is collected without your intervention, but collecting garbage still takes time.
- Garbage collection is not a guarantee that memory will always be available for new objects. It solves many but not all the memory allocation problems

Nested Classes and Interfaces

- Classes and interfaces can be declared inside other classes and interfaces, either as members or within blocks of code.

Static Nested Classes/Interfaces — Overview

- A nested class/interface which is declared as `static` acts just like any non-nested class/interface, except that its name and accessibility are defined by its enclosing type.
- Static nested types are members of their enclosing type
  - They can access all other members of the enclosing type including the private ones.
  - Inside a class, the static nested classes/interfaces can have private, package, protected or public access; while inside an interface, all the static nested classes/interfaces are implicitly public.
  - They serve as a structuring and scoping mechanism for logically related types

Static Nested Classes/Interfaces (cont.)

- Static nested classes
  - If a class is nested in an interface, it's always static (omitted by convention)
  - It can extend any other class, implement any interface and itself be extended by any other class to which it's accessible
  - Static nested classes serve as a mechanism for defining logically related types within a context where that type makes sense.

```
public class BankAccount {
   private long number;       //account number
   private long balance;      //current balance

   public static class Permissions {
        public boolean canDeposit, canWithdraw, canClose;
   }
// . . .
}
```

  - Code outside the `BankAccount` class must use `BankAccount.Permissions` to refer to this class

```
BankAccount.Permissions perm = acct.permissionsFor(owner);
```

- Nested interfaces
  - Nested interfaces are always static (omitted by convention) since they don't provide implementation

Non-static Classes — Inner classes

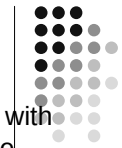
- *Inner classes* are associated with instances of its enclosing class.

```
public class BankAccount {
   private long number;    // account number
   private long balance;   // current balance
   private Action lastAct; // last action performed

   public class Action {
      private String act;
      private long amount;
      Action(String act, long amount) {
         this.act = act;
         this.amount = amount;
      }
      public String toString() {
         //identity our enclosing account
         return number + ": " + act + " " + amount;
      }
   }

   public void deposit(long amount) {
      balance += amount;
      lastAct = new Action("deposit", amount);
   }
   public void withdraw(long amount) {
      balance -= amount;
      lastAct = new Action("withdraw", amount);
   }
   // . . .
}
```

Inner classes (cont.)



- When an inner class object is created, it MUST be associated with an object of its enclosing class. Usually, inner class objects are created inside instance methods of the enclosing class. When this occurs, the current enclosing object `this` is associated with the inner object by default.

```
lastAct = this.new Action("deposit", amount);
```

  - When `deposit` creates an `Action` object, a reference to the enclosing `BankAccount` object is automatically stored in the `Action` object.
  - Using the saved reference, the inner-class object can refer to the enclosing object's fields directly by their names. The full name will be the enclosing object `this` preceded by the enclosing class name

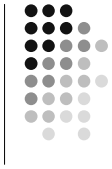
```
return BankAccount.this.number + ": " + act + " " + amount;
```

  - Any enclosing object can be substituted for `this.`

    Example: suppose a new method named `transfer` is added

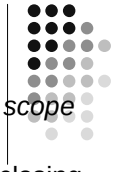
```
public void transfer(BankerAccount other, long amount) {
   other.withdraw(amount);
   deposit(amount);
   lastAct = this.new Action("transfer", amount);
   other.lastAct = other.new Action("transfer", amount);
}
```

Inner classes (cont.)



- The enclosing class can also access the private members of its inner class, but only via explicit reference to an inner class object.
- An object of the enclosing class need not have any inner class objects associated with it, or it could have many.
- An inner class acts as a top-level class except that it can't have static members (except for final static fields).
- Inner classes can also be extended.

Inheritance, Scoping and Hiding



- All members declared within the enclosing class are said to be in *scope* inside the inner class.
- An inner class's own fields and methods can hide those of the enclosing object. Two possible ways:

1). A member with the same name is declared in the inner class

- Any direct use of the name refers to the version inside the inner class

```
class Host {
   int x;
   class Helper {
      void increment() {int x=0; x++;}
   } }
```

- Access to the enclosing object's members needs be preceded by `this` explicitly

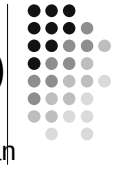
2). A member with the same name is inherited by the inner class

- The direct use of the name is not allowed

```
class Host {
   int x;
   class Helper extends Unknown {    //Unknown class has a field x
      void increment() {x++;}
   }
}
```

- Use `enclosingClassName.this.name` to refer to the version in the outer class
- Use `this.name` or `super.name` to refer to the version in the inner class

Inheritance, Scoping and Hiding (cont.)

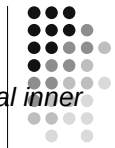


- A method within an inner class which has the same name as an enclosing method hides all overloaded forms of the enclosing method, even if the inner class itself does not declare those overloaded forms.

```
class Outer {
   void print() {}
   void print(int value) {}

   class Inner {
      void print() {}
      void show() {
         print();
         Outer.this.print();
         print(1); //INVALID: no Inner.print(int)
      }
   }
}
```

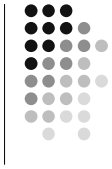
Local Inner Classes



- You can define inner classes in code blocks. They are called *local inner classes.*
  - They are NOT members of the class which contains the code but are local to that block, as a local variable.
  - They are completely inaccessible outside of the block.
  - Only one modifier is allowed—`final`—which makes them unextendable
  - It can access all of the `final` variables and method parameters that are in scope where the class is defined.

```
public static Iterator walkThrough(final Objects[] objs) {
   class Iter implements Iterator{
      private int pos = 0;
      public boolean hasNext() {
         return (pos < objs.length);
      }
      public Ojbect next() throws NoSuchElementException {
         if (pos >= objs.length)
            throw new NoSuchElementException();
         return objs[pos++];
      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
   }
   return new Iter();
}
```

Anonymous Inner Classes

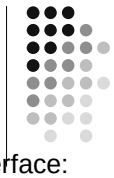


- You can declare *anonymous classes* that extend a class or implement an interface. This type of classes are defined at the same time they are instantiated with `new`.

```
public static Iterator walkThrough(final Objects[] objs) {
   return new Iterator() {
      //same code as those inside the body of the Iter class
   };
}
```

  - Anonymous classes can't have explicit `extends` or `implements` clause
  - The type specified to `new` is the supertype of the anonymous class
  - Anonymous classes can't have explicit constructors declared

Nesting Inside Interfaces

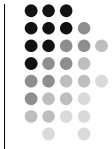


- Reasons for using nested classes and interfaces inside an interface:
  - Associate types that are strongly related to an interface inside that interface

```
Interface Changeable {
    class Record {
        public Object changer;
        public String changeDesc;
    }
    Record getLastChange();
    // . . .
}
```

  - To define a (partial or complete) default implementation for that interface. A class implementing the interface can choose to extend the default implementation or simply follow it.
- Any class or interface nested inside an interface is public and static

Modifiable Variables in Interfaces



- If you need shared, modifiable data in an interface, then an inner class is a simple way of achieving this:
  - Declare an inner class whose fields hold the shared data
  - The class's methods provide access to the data
  - Maintain a reference to an instance of that class

```
Interface SharedData {
    class Data {
        private int x = 0;
        public int getX() { return x; }
        public void setX(int newX) { x = newX; }
    }
    Data data = new Data();
}
```