Java II

Walter Wesley
UCSD Extension
Information Technology and
Software Engineering

Module 4

Defining a Class

Default Initialization

Constructors

Constructor Overloading

Using the Debugger

Setters and Getters

Javadoc Comments

Software Reuse via Inheritance

Polymorphism (Dynamic Binding)

ArrayLists

Working with Strings

Working with Files

Module 4

Exceptions

The Random Class

Defining a Class

- The general syntax for a *class definition* is as follows:

```
public class Person
{
  .
  .
  .
}
```

- Only one *public class definition* can exist within any given *.java* file.

Defining a Class

- Once the class ***Person*** has been defined, a ***Person object reference*** can be declared as follows:

  Person person;

- A ***Person*** object can then be instantiated and assigned to the ***object reference*** as follows:

  person = new Person();

- Alternatively, an ***object reference*** can be declared and assigned (initialized with) the value of an instantiated object all at once with one statement as follows:

  Person person = new Person();

- Object fields (instance variables) are declared at the class scope (outside of any methods).

Defining a Class

- ***Object fields*** are ***declared*** as follows:

```
class Person
{
  // Object Fields (note that these should always be declared private!)
  private String name;
  private int age;
  private String ssiNumber;
  .
  .
  .
}
```

- Any object created from the above Person class will have its own name, age, and ssiNumber inside of it.

Default Initialization

- Object fields that are not explicitly initialized as part of their declaration will automatically be initialized by the compiler with default values.
- The default initialization values used by the compiler are:
  - *null* for object references.
  - *0* for numbers.
  - *false* for booleans.
- Accordingly, objects instantiated from our *Person* class will have their *name*, *age*, and *ssiNumber* object fields initialized with values of *null*, *0*, and *null*, respectively.

Constructors

- The default initialization values are rarely the values you want for your object fields.
- Often, it is at the time of object instantiation that you know what values you want your object fields to be initialized with.
- For this purpose, there is a special method called a ***constructor*** that is ***responsible for initializing your object fields*** with the values that you pass in as arguments.
- There are ***two characteristics*** that identify a method as being a constructor:
  a. ***Constructors do not have a return type.***
  b. ***The name of a constructor is identical to its classname.***
- Constructors are typically used to ensure that object is ***fully formed*** and in a ***reliably usable state***.

Constructor Overloading

- It is possible to have more than one constructor.
- Multiple constructors are differentiated based upon the number of arguments and the type of arguments.
- Consider the following:

```
public Person(String name, int age, String ssiNumber)
{         // A Person constructor that initializes all object fields.
  this.name = name;
  this.age = age;
  this.ssiNumber;
}

public Person(String name)
{         // A Person constructor that initializes only the “name” object field.
  this.name = name;
}
```

Using the Debugger

- DrJava, as is the case with most full-featured IDEs, includes a facility known as a ***debugger***.
- ***Debuggers*** enable a software developer to set breakpoints, locations at which the program stops while running, thereby allowing the developer to inspect the state of the running program.
- With DrJava, you can type a variable name into the ***name*** field within the ***Watches*** pane, and you will then see the current ***value*** and ***type*** of that variable.
- Alternatively, you can use the ***Interactions*** pane to Java statements to either ***show or manipulate program data***.

Using the Debugger

- Clicking on the ***Resume*** button will reactivate the execution of the program.
- Clicking on the ***Step Into*** button will follow the executing program into the body of a method invocation.
- Clicking on the ***Step Over*** button will follow the executing program in and out of a method invocation in one discrete step.
- Clicking on the ***Step Out*** button will complete the remaining statements within the current method body and stop at the statement immediately following the invocation of the current method.
- You should practice using the ***Debugger***, as it is an extremely powerful and useful tool.

Setter and Getters

- Let us assume that you instantiated a Person object as follows:

  Person fred = Person("Frederick");

- In this case, the ***fred*** reference variable refers to a ***Person*** object, for which the ***name*** object field has been initialized with the string ***"Frederick"***.
- Meanwhile, since the object fields ***age*** and ***ssiNumber*** were not explicitly initialized, they were implicitly initialized by the compiler to be ***0*** and ***null***, respectively.
- It makes sense to construct a ***Person*** object in this way, provided that we do not know ***fred***'s age or social security number at the time that we need to create him.
- However, ***when we finally do know*** his age and social security number, ***how do we set those object fields?***

Setter and Getters

- The convention is Java is to create “set” methods and “get” methods (also called setters and getters, or modifiers and accessors), that selectively allow for the value of a given object field to be modified or accessed, respectively.
- Consider the following (remember ***age*** is a ***private instance variable***):

```
public void setAge(int age)
{        // Setter method sets age to incoming argument value.
  this.age = age;
}

public int getAge()
{        // Getter method gets value of private object field age.
  return age;
}
```

Setter and Getters

- Therefore, assuming that the appropriate setters and getters have been defined, from the interactions pane we should be able to do the following:

  Person fred = Person("Frederick");

  fred.setAge(28);

  fred.setSSINumber("314159265");

- Now, here's something to think about..., does someone actually have that social security number, and if so, are they aware of its significance?

Javadoc Comments

- So far, we have been using two types of comments, single-line comments (//...) and multi-line comments (/*... */).
- There is another very important type of comment known as a ***Javadoc comment***.
- ***Javadoc*** is a special java utility that scans through your source code (.java files) and looks for ***special tags*** that help it build an ***HTML*** formatted documentation file.
- Remember the ***Java API Specification*** that you have been exploring so assiduously? That was created by the ***Javadoc*** utility! Since the output of the ***Javadoc*** utility is an ***HTML*** document, you can ***easily view that document with any web browser***.
- Just as the core Java class developers did with their classes, you can ***document your own classes***, injecting the appropriate ***tags*** in the appropriate places so that the ***Javadoc*** utility (or your IDE, if it supports it) can ***generate an HTML API document***.

Javadoc Comments

- Documentation of your class, providing a purpose and description, is accomplished as follows:

```
/**
 * Class that is indisputably the coolest class ever
   composed!
 * @author  Duane Wesley
 */
```

- The ***/**** indicates the beginning of the ***Javadoc comment***, and the ****/*** indicates the end. You may want your comments to be more informative and humble than the one shown above. ;-)
- The ***@author*** is a ***special tag*** that indicates that ***the author's name immediately follows***.

Javadoc Comments

- The methods within your class definition can be documented as follows:

```
/**
 * Method that sets a score within the gameScore array
 * @param index the index to set the score at
 * @param newScore the new score to use
 * @return true if success, else false
 */
```

- ***@param*** is a ***special tag*** that indicates that ***a parameter description immediately follows***. The order of appearance should match the parameter list order.
- ***@return*** is a ***special tag*** that indicates that ***a return value description immediately follows***.

Javadoc Comments

- Many IDEs facilitate the invocation of the ***Javadoc*** utility. In the case of ***DrJava***, you merely click on the ***Javadoc button***.
- Alternatively, you can manually execute the ***Javadoc*** utility at a command prompt from within a console window.
- At a command prompt, you can generate ***HTML*** documentation for all the classes within the current directory by typing..

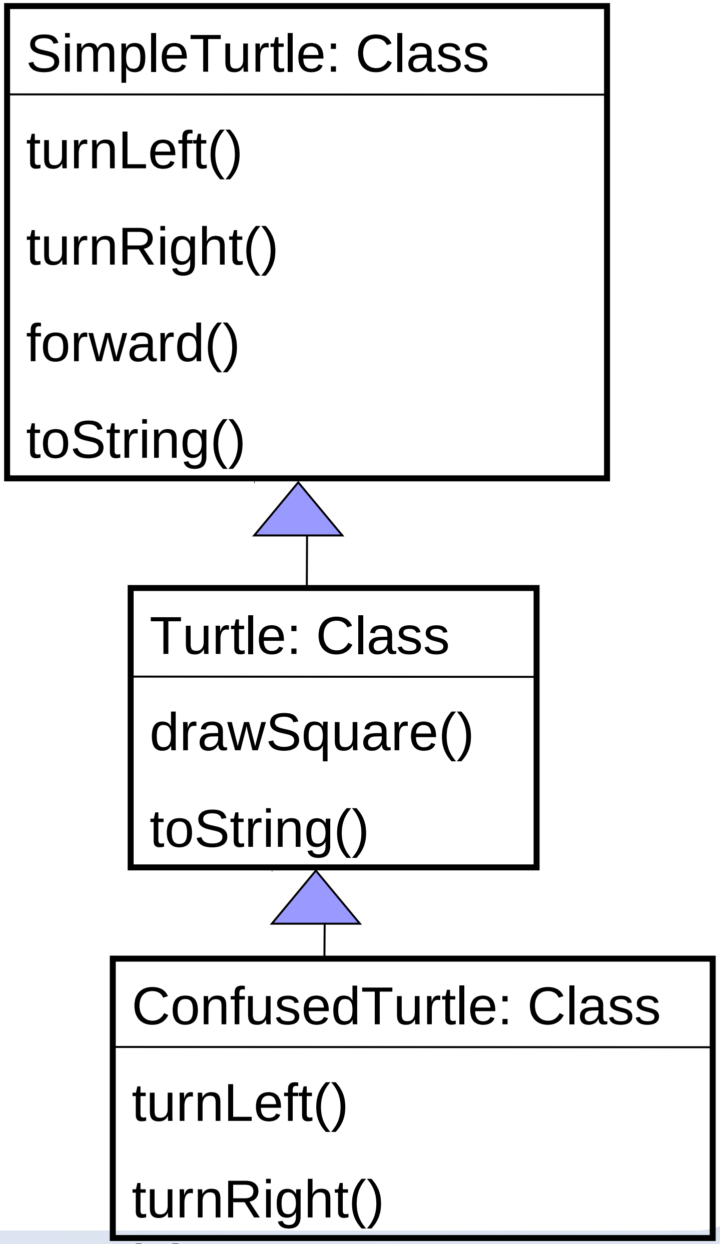
  javadoc *.java

- If you only want to generate ***HTML*** documentation for a single class, then you can specify the ***.java*** file without the wildcard designation:

  javadoc Person.java

Software Reuse via Inheritance

- Resolving Methods
  - When a method is invoked (called) on an object
    - The class that created the object is checked
      - To see if it has the method defined in it
        - If so it is executed
        - Else the parent of the class that created the object is checked for the method
        - And so on until the method is found
  - Super means start checking with the parent of the class that created the object



Software Reuse via Inheritance

- Creating Classes with Turtles
  - Overriding our basic object Turtle to make a Confused (or Drunken) Turtle
  - Create a class ConfusedTurtle that inherits from the Turtle class
    - But when a ConfusedTurtle object is asked to turn left, it should turn right
    - And when a ConfusedTurtle object is asked to turn right, it should turn left

Software Reuse via Inheritance

- Inheriting from a Class
  - To inherit from another class
    - Add extends ClassName to the class declaration

```
public class ConfusedTurtle extends Turtle
{
}
```

  - Save in ConfusedTurtle.java
  - Try to compile it

Software Reuse via Inheritance

- Compile Error?
  - If you try to compile ConfusedTurtle you will get a compiler error
    - Error: cannot resolve symbol
    - symbol: constructor Turtle()
    - location: class Turtle
- Why do you get this error?

Software Reuse via Inheritance

- Inherited Constructors
  - When one class inherits from another all constructors in the child class will have an implicit call to the no-argument parent constructor as the first line of code in the child constructor
    - Unless an explicit call to a parent constructor is there as the first line of code...

      super(paramList);

Software Reuse via Inheritance

- Why is an implicit call to super added?
- Fields are inherited from a parent class
  - But fields should be declared private
    - Not public, protected, or package visibility
      - Lose control over field at the class level then
  - But then subclasses can't access fields directly
  - How do you initialize inherited fields?
    - By calling the parent constructor that initializes them...

      super(paramList);

Software Reuse via Inheritance

- Explanation of the Compile Error
  - There are no constructors in ConfusedTurtle
    - So a no-argument one is added for you
      - With a call to super();
    - But, the Turtle class doesn't have a no-argument constructor
      - All constructors take a world to put the turtle in
  - So we need to add a constructor to ConfusedTurtle
    - That takes a world to add the turtle to
      - And call super(theWorld);

Software Reuse via Inheritance

- Add a constructor that take a World

```
public class ConfusedTurtle extends Turtle
{
  /**
   * Constructor that takes a world and
   * calls the parent constructor
   * @param theWorld the world to put the
   * confused turtle in
   */
  public ConfusedTurtle(World theWorld)
  {
    super (theWorld);
  }

}
```

Software Reuse via Inheritance

- Try this Out
  - Compile ConfusedTurtle
    - It should compile
    - It should act just like a Turtle object
  - How do we get it to turn left when asked to turn right? And right when asked to turn left?
    - Use super.turnLeft() and super.turnRight()
    - super is a keyword that means the parent class

Polymorphism (Dynamic Binding)

- Means many forms
- Allows for processing of an object based on the object's type
- A method can be declared in a parent class
  - And redefined (overridden) by the child classes
- Dynamic or run-time binding will make sure the correct method gets run
  - Based on the type of object it was called on at run time

Polymorphism (Dynamic Binding)

```
/**
 * Method to turn left (but confused turtles
 * turn right)
 */
public void turnLeft()
{
  super.turnRight();
}

/**
 * Method to turn right (but confused turtles
 * turn left)
 */
public void turnRight()
{
  super.turnLeft();
}
```

Polymorphism (Dynamic Binding)

- Try out ConfusedTurtle

```
> World earth = new World();
> Turtle tommy = new Turtle(earth);
> tommy.forward();
> tommy.turnLeft();
> ConfusedTurtle bruce = new ConfusedTurtle(earth);
> Turtle someTurtle = bruce;
> bruce.backward();
> someTurtle.turnLeft();  // This is a polymorphic call
> bruce.forward();
> tommy.forward();
> tommy.turnRight();
> someTurtle.turnRight();   // This is a polymorphic call
```

Polymorphism (Dynamic Binding)

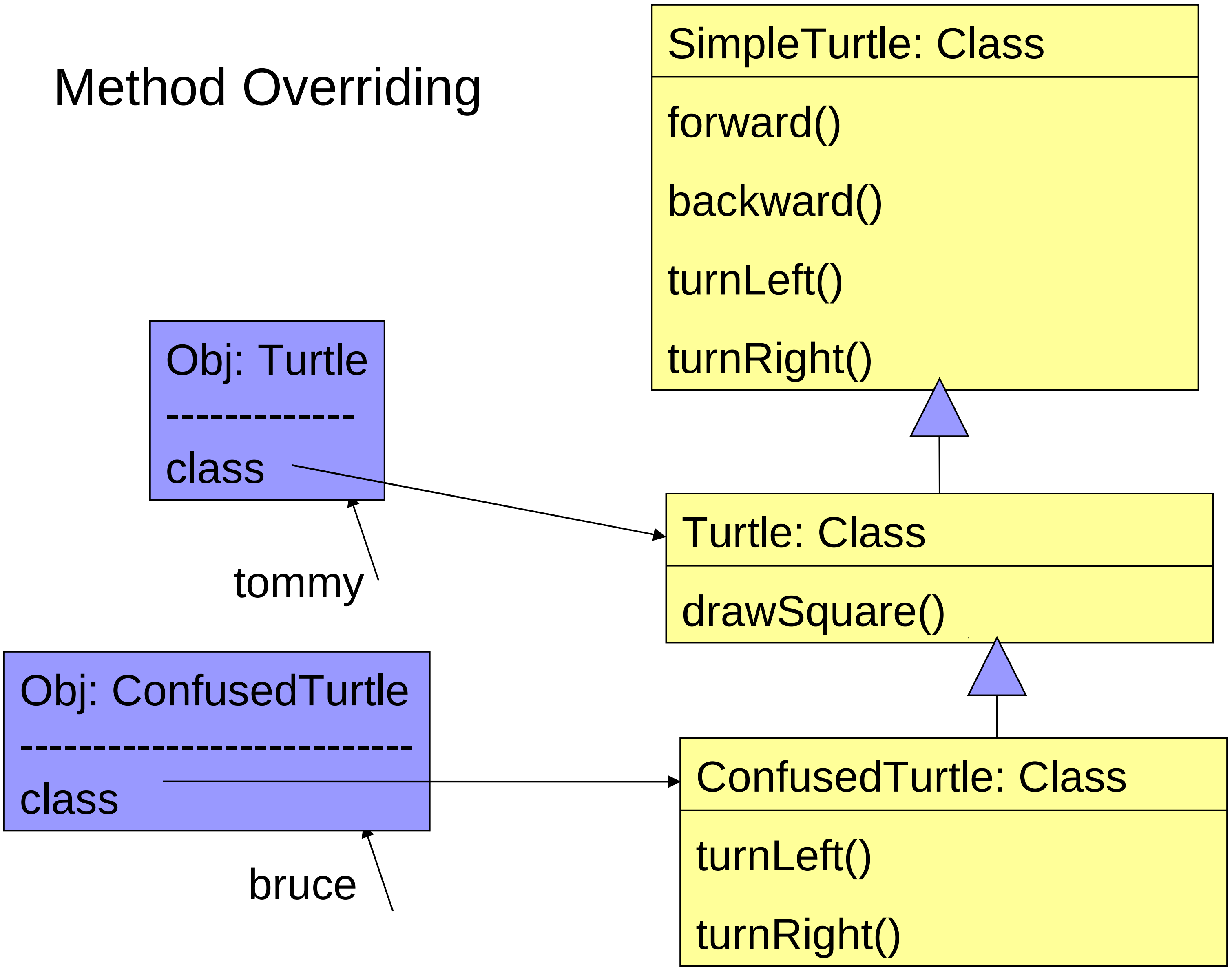
- Override Methods
  - Children classes inherit parent methods
    - The confused turtle knows how to go forward and backward
      - Because it inherits these from Turtle
  - Children can override parent methods
    - Have a method with the same name and parameter list as a parent method
      - This method will be called instead of the parent method
        - Like turnLeft and turnRight

Polymorphism (Dynamic Binding)

- What is Happening?
  - Each time an object is asked to execute a method
    - It first checks the class that created the object to see if the method is defined in that class
      - If it is it will execute that method
      - If it isn’t, it will next check the parent class of the class that created it
        - And execute that method if one if found
        - If no method with that name and parameter list is found it will check that classes parent
        - And keep going till it finds the method

Polymorphism (Dynamic Binding)

Method Overriding



ArrayLists

- So far, we have been using Java arrays in order to manage a set of elements of the same type (*ints*, *doubles*, *Strings*, or whatever).
- *ArrayList* is an extremely useful class. *ArrayList* objects are similar to regular Java arrays, except that *ArrayList objects resize themselves dynamically as needed*, absolving you of the responsibility of dimensioning them!
- You should review the API for this class within the Java API Documentation document. Specifically, you should look at the *add*, *get*, *set*, *size*, and *remove* methods.
- Enter the statements on the following slide within the DrJava *Interactions* pane.

ArrayLists

```
> World world = new World()
> import java.util.*
> ArrayList pond = new ArrayList()
> Turtle fred = new Turtle(world);
> Turtle sally = new Turtle(world);
> Turtle george = new Turtle(world);
> pond.add(fred);
> pond.add(sally);
> pond.add(george);
> fred.turn(90)
> george.turn(270)
> for(Object turtle : pond)
{
  ((Turtle) turtle).forward(200); // We cast by putting (Turtle) in front of turtle.
}
```

ArrayLists

- One advantage to using an ***ArrayList*** is that we do not need to manage an explicit index; we merely add and iterate, without caring about where we might be within the collection, and without worrying about over-running any bounds (with an ***ArrayList***, there are no bounds!).
- A disadvantage is that a plain ***ArrayList*** stores ***Objects***. This means that when you add something to ***ArrayList***, you lose the information about what the ***Object*** really is. Once you add something to a plain ***ArrayList***, it is no longer a ***Person***, or a ***Turtle***, or a ***String***, but rather it is only an ***Object***. That is why when you retrieve an ***Object*** from a plain ***ArrayList***, you must ***cast*** it to the type that you know it is before you invoke methods upon it. That is the reason for the statement...

  ((Turtle) turtle).forward(200);

- We ***cast*** from ***Object*** to ***Turtle*** by putting ***(Turtle)*** in front of ***turtle***.
- Because we are ***casting downward in the inheritance hierarchy***, this is known as ***downcasting***.

ArrayLists

- Better than using a plain ***ArrayList***, we can make use of a ***generic***.
- ***Generics*** are a feature that were added to Java with Version 5 of the language.
- ***Generics*** are a mechanism by which you can bind a type to a class or method. In this way, you can develop structures that operate with any type that you specify—that's why they call them ***generics***.
- ***Generics*** will be covered in greater detail in Java III, but in our case here we can improve our use of ***ArrayList*** by using the ***generic*** syntax to bind to the ***Turtle*** class:

  ```
  > ArrayList<Turtle> turtlePond = new ArrayList<Turtle>()
  ```

- With the above ***generic*** declaration, it is no longer necessary to cast from ***Object*** to ***Turtle***. So instead of...

  ```
  > ((Turtle) turtle).forward(200);
  ```

  we have

  ```
  > turtle.forward(200);
  ```

ArrayLists

```
> World world = new World()
> import java.util.*
> ArrayList<Turtle> turtlePond = new ArrayList<Turtle>()  // This is the generic
    version.
> Turtle fred = new Turtle(world);
> Turtle sally = new Turtle(world);
> Turtle george = new Turtle(world);
> pond.add(fred);
> pond.add(sally);
> pond.add(george);
> fred.turn(90)
> george.turn(270)
> for(Turtle turtle : turtlePond)
{
  turtle.forward(200);   // No casting is necessary!
}
```

Working with Strings

- In Java, *Strings* are first class objects.
- *Strings* provide a representation of a series of characters, and therefore are useful tools for manipulating text.
- However, *Strings* are *immutable*; once a *String* has been formed, it cannot be changed!
- If you have a *String* that is close to but not exactly what you want, you need to create a new *String*.
- Fortunately, the *String* class offers a wide variety of utility methods that assist you in working with *Strings*. Many of these methods allow you to create new *String* objects based upon pre-existing *String* objects.
- You should browse through the API documentation, and familiarize yourself with the riches of the *String* class.

Working with Files

- An important aspect of any language is how to read from and write to files.
- As part of reading and writing data to a file, it is invariably more efficient to provide a buffer between the program and the output device.
- The ***java.io package*** has classes that deal with the buffering issues in a very elegant, modular way.
- The next two slides present code examples of how to read from and write to files, respectively.

Working with Files

```
// Reading from a file
import java.io.*;

String fileName = "full pathname to file";
BufferedReader buffReader = new FileReader(fileName);
String aLine = null;

while((aLine = buffReader.readLine()) != null)
{
  System.out.println(aLine);
}

buffReader.close();
```

Working with Files

```
// Writing to a file
import java.io.*;
String fileName = "full pathname to file";
BufferedWriter buffWriter = new FileWriter(fileName);
String aLine = "I would like to have a cookie!";

buffWriter.write(aLine);
buffWriter.newLine();
buffWriter.write(aLine);
buffWriter.newLine();
buffWriter.write(aLine);
buffWriter.newLine();

buffWriter.close();
```

Exceptions

- There is a problem with the previous two slides. It is possible for an ***I/O Exception*** to be thrown from the ***BufferedReader readLine*** method or from the ***BufferedWriter writeLine*** method.
- ***Exceptions*** are a way of indicating that a serious error has been encountered.
- ***Exception handling*** is a means by which thrown ***exceptions*** can be ***caught*** and ***handled***. Handling usually involves attempting to resolve the problem somehow.
- Method invocations that can possibly throw an ***exception*** are enclosed within a ***try block***.
- Code that should be executed when an exception has been thrown is enclosed within a ***catch block***.
- Code that should be executed under all circumstances is enclosed within a ***finally block*** (the ***finally block*** is optional).
- The following two slides demonstrate the file reading and writing code with ***exception handing*** code included.

Exceptions

```
// Reading from a file
import java.io.*;
String fileName = "full pathname to file";
BufferedReader buffReader = null;
try {  // beginning of try block
        buffReader = new FileReader(fileName);
        String aLine = null;

        while((aLine = buffReader.readLine()) != null)
        {
     System.out.println(aLine);
        }
}  // end of try block
catch (FileNotFoundException e) {
        System.out.println("File " + fileName + " not found!");
}
catch (Exception e) {
        e.printStackTrace();
}
finally {
        if (buffReader != null) buffReader.close();
}
```

Exceptions

```
// Writing to a file
import java.io.*;
String fileName = "full pathname to file";
BufferedWriter buffWriter = null;
try {
        buffWriter = new FileWriter(fileName);
        String aLine = "I would like to have a cookie!";
        buffWriter.write(aLine);
        buffWriter.newLine();
}
catch (Exception e) {
        e.printStackTrace();
}
finally {
        if (buffWriter != null) buffWriter.close();
}
```

The Random Class

- The *Random* class is a very useful class. You can do some really fun things with it!
- Interestingly, since computers are deterministic machines, they cannot manifest true randomness. They can only simulate it!
- The following demonstrates how to use the *Random* class:

```
import java.util.Random;
Random random = new Random();

int someInt = random.nextInt(101);  // from 0 to 100
System.out.println(someInt);

double someDouble = random.nextDouble(); from 0 to 1
System.out.println(someDouble);
```

The Random Class

- Imagine how you could use the ***Random*** class to implement a ***DrunkenTurtle*** class. You could ***override*** the turn and forward methods ***in a way that randomizes*** the direction and the length of walking!