Introduction to Java for z/OS Applications Developers - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Describe how to determine the available version of Java on your system

2. Code server-side Java programs for z/OS that:

    * Define and work with data items of all Java primitive types
    * Use operators with data items and literals in expressions
    * Create packages and the underlying directory structure to support this
    * Declare and define classes, including fields, methods, initializers, and constructors
    * Define subclasses and reference methods and fields using 'this' and 'super'
    * Accept command line arguments and process them
    * Accept and work with system parameters
    * Handle arrays of various types
    * Use complex logic structures and labels
    * Use and implement abstract classes and wrapper classes
    * Use regular expressions
    * Convert data between formats, explicitly and implicitly

3. Locate and use information available in the Sun-supplied online Java documentation

4. Handle exceptions, including try, catch, and finally blocks, and creating your own exception classes

5. Write and access data in HFS / zFS files and classic MVS data sets in ASCII, Unicode, and EBCDIC

6.

Notes: This course includes the latest version of Java available for z/OS, Technololgy Edition 6.

The prerequisite is basic concepts and commands of z/OS UNIX,including the use of oedit and obrowse or vi for text file creation, viewing, and modification.

Introduction to Java for z/OS Applications Developers - Topical Outline

Day One



Day Three, continued



Day Three, continued

Reading from HFS Files
- Reading fields and records
- Classes for file input
- InputStream classes for file input
- Reclaiming data
- Instantiating Objects
  - Using the FileInputStream Class for File Input
  - Using the FilterInputStream Class for File Input
  - Using the BufferedInputStream Class for File Input
  - Using the DataInputStream Class for File Input
- Reader classes for file input
  - Using the BufferedReader Class for File Input
  - Using the InputStreamReader Class for File Input
  - Using the FileReader Class for File Input
- A Lesson in Scope
- Computer Exercise: Add, Display, Update, and Delete Files ..... 407

Day Four

Numbers and and the java.math package
- Looking at Unit Price
- Precision
- Rounding
- The MathContext class
- The BigDecimal class
- The BigInteger class
- The java.math package
- Computer Exercise: Using BigDecimal .....

Dates, Times, and Formatting

NOTE: This course is designed for Java programs to be coded, compiled, and tested on the z/OS platform. In certain situations it might be more productive to code and compile on the workstation, then, when there is a clean compile to either 1) upload the .java file and compile on the mainframe or 2) upload the .class file directly to the mainframe. In either case, testing should be done under z/OS UNIX.

# Section Preview

- **Introduction to Java**
  - **Java**
  - **Java - Versions**
  - **Java - Editions and Tool Kits**
  - **Java - Basic Programming**
  - **Java Programming Approach in this Course**
  - **Setting Up for the Labs (Machine Exercise)**

This page intentionally left almost blanki.

# Java

❒ **Java is a programming language**

- ♦ **Created and controlled by Sun Microsystems, Inc.**
  - ✗ **With the motto "Code once, run anywhere" and the mantra "code reuse"**

❒ **The central theme of Java, however, is that it is object oriented, so it deals with terms and concepts such as**

- ♦ **Class - a model that describes the common parts of a bunch of "thingys" that we care about (for example: customers, items in inventory, our employees)**
  - ✗ Class definitions specify the fields and methods (data and programs) that make up a class
- ♦ **Object - each object is a particular thingy; called an instantiation of the class (for example: "The Hobby Cat", "Fiberglass Cups", "Theopholus Whifflepoof")**
- ♦ **Fields - the data items associated with objects in a class (also called "variables", "properties", and "attributes") (for example, company name, quantity on hand, employee id)**
  - ✗ Some fields may describe the class instead of individual objects
- ♦ **Methods - code routines for each class that may be invoked against objects in the class (for example, calculate a discount, deduct from inventory, assign a worker to a department)**
  - ✗ Some methods apply against the class or are abstract
- ♦ **Interface - a collection of related method prototypes (for example, looking up an address or location, printing a label)**

# Java, 2

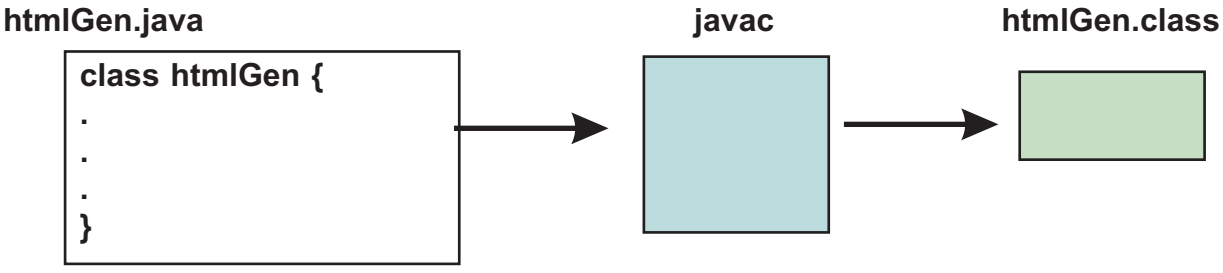
❒ **Write once, run anywhere - How does it do that?**

❒ **Like any computer program, a Java program is coded as a text file**

- **Which must be named *class-name*.java where *class-name* is the name of the main or only class defined in the program**
- **Every program defines one or more classes - specifies the data and methods available for all objects belonging to each class**
  - ✗ **Subclasses may be defined in the same or a separate source program**

❒ **Next, a java program is compiled using the z/OS UNIX line command** javac

- **Assuming your program has no errors, the compile produces a file named *class-name*.class**
  - ✗ The contents of this file are Java bytecode: pseudo instructions and data defined by Sun's Java specification
  - ✗ For every operating system that supports Java, there is a javac compiler that converts text coded on that platform to Java bytecode



# Java, 3

❒ **To run a Java program, you launch it with the** java **command from a z/OS UNIX command line (or, possibly, a script):**

**java** *class-name*

- ♦ **This loads a program called the Java Virtual Machine (JVM), loads the named class, and begins running the program**

❒ **The JVM is the key to the promise "Write once, run anywhere"**

❒ **All the major software operating systems creators provide a JVM that interprets Java bytecode, using each machine's unique hardware instructions and data formats to produce the result dictated by the Java language**

❒ **So each platform has a Java compiler and a JVM**

❒ **The compiler produces Java bytecode that runs on any JVM - so Java source code need only be compiled once and it can run on any platform - in theory**

- ♦ **There are some situations where it doesn't quite work, for example:**

  ✗ If you compile on a current version of Java and try to run on the JVM from an older version - the older version might not support all the features you used in your new code

  ✗ If you use features, such as graphical image presenation, not available on the system running the program, such as z/OS

- ♦ **But these are mostly minor issues**

# Java - Versions

❒ **Like any living language, Java has gone through changes, and in the literature you will see these terms**

- **Java 1.0 - the first significant release; 1996**
- **Java 1.1 - 1997**
- **Java 1.2 - late 1998; this is also referred to as "Java 2"**
- **Java 1.3 - 2000; still called "Java 2"**
- **Java 1.4 - late 2001; still called "Java 2"**
- **Java 5.0 - September 2004; still called "Java 2"**
- **Java SE 6 - December 2006; from the Java website: "At this release, the platform name has changed from J2SE™ to Java™ SE. The official name is Java™ Platform, Standard Edition 6."**

  ✗ Under this platform, Sun delivers two major products:

  ➢ Java SE Development Kit 6, abbreviated JDK 6

  ➢ Java SE Runtime Environment 6, abbreviated JRE 6

❒ **You can find out what version of java you are running by issuing one of these commands on the omvs or telnet command line:**

**java -version**
**java -fullversion**
**javac -J-version**

Let's Do It!

Computer Experiment: Find out the level of Java you are running (postpone if not possible)

Get to omvs or a telnet command line, and issue the above commands.

# Java - Editions and Tool Kits

- With different editions, Sun has provided a variety of tools and tool kits to support Java, often changing the name and then changing it back; you will see these terms
  - J2SE - Java 2 Standard Edition - most common in corporate environments (now called "Java SE", see previous page)
  - J2EE - Java 2 Enterprise Edition - extends J2SE by adding features such Enterprise Java Beans (EJBs)
  - J2ME - Java 2 Mobile Edition
  - JRE - Java Runtime Environment
  - JDK - Java Developer's Kit (also known as SDK)
  - SDK - Software Developer's Kit (includes a JRE)
  - Java BluePrints - pre-developed collections of guidelines, patterns, and code (see http://java.sun.com/reference/blueprints/)
- In this course we focus on the essentials of the Java language, which are the same on all platforms
  - With emphasis on coding, compiling, and running on the z/OS platform
    - ✗ Discussing areas where there may be issues or differences
- IBM puts out what it calls Technology Editions which correspond, roughly, to Sun's Java SE; IBM's WebSphere Application Server (WAS) provides J2EE features and support; at time of publication of this course, the latest version of Java supported on z/OS is 6.0

# Java - Basic Programming

❒ **Here is a small Java program**

```
❶    // Declaration of class Sample1
❷     class Sample1 {   // Definition of class Sample1...
❸
❹      // Definition of method "main"
❺      public static void main(String[] args) {

       // Declaration of local variable "total"
❻        int total = 0;

       // Logic of "main"
❼        for (int counter = 1; counter < 10; counter++) {
           total += counter;
❽          System.out.println("counter = " + counter);
           System.out.println("    total = " + total);
         } // end of loop

❾      /* print the final result */

❿        System.out.println("\nGrand total = " + total);

       } // end of method "main"

     }   // end of class definition Sample1
```

**Notes**

**❶ Comments come in various styles; the "//" begins a comment that says "the rest of the line is a comment"**

**❷ A class is defined beginning with zero or more class modifiers (for this simple situation, none is required; more later, as needed), the word "class", the name of the class, and an opening brace "{"; [there is a corresponding closing brace later, to designate the end of the class definition]**

**❸ Blank lines may appear anywhere**

# Java - Basic Programming

## Notes, continued

❹ A class may have many methods and variables; the class that drives a Java application must have a method named "main" - this is where execution of the application will begin

❺ A method is defined beginning with zero or more method modifiers (for the top level method, "public" and "static" are required), the type of data the method returns (or "void" if it returns no data), the method name, an open parenthesis followed by a list of arguments the method expects and the name you want to give each argument (for "main" methods, as in the example, it is always String[] and usually a name of "args"), a closing parenthesis, one or more spaces and an open brace

❻ variables are declared with the syntax data-type, name, and, optionally, an initial value; all Java statements end with a semi-colon

❼ details of statements are covered as we progress

❽ The System.out.println method is how you write a string to stdout (details later); but observe the quoted string, the "+" and the variable name, concatenating the literal string and the int variable

observe also that the int variable is being converted to a String automatically here

❾ Here is another style of comment, with opening and closing bounds

❿ Observe the "\n" - an escape character indicating "start a new line"

# Java Programming Approach In This Course

- **The course author is not a fan of "Hello World" labs, because they are not at all real world**

- **At the same time, a lot of concepts have to be covered before we can demonstrate using files or databases from a Java program**

  - **Working with such data is considered to be the hallmark of a "real" application**

- **So we content ourselves by using the System.out.println method demonstrated in the example above as a way to display footprints and breadcrumbs**

  - **"Footprints" being messages indicating where we are in some code**

  - **"Breadcrumbs" being the content of variables**

  - **Note that** System.out.println() **forces a new line after writing out the data; you may also use** System.out.print() **which leaves the next location in the output stream as the target**

    - ✗ In both functions, the escape sequence "\n" introduces an additional line break

Computer Exercise: Setting up for the Labs

At this point, you should get to a z/OS UNIX command line and issue the following commands:

```
mkdir J510              <-- create a new directory
cd J510                 <-- make it the current working directory
umask 000               <-- turn off masking of permission bits
pax -r -f /_________/J510files.pax.Z      <-- unwind class files
ls                      <-- see what files are there to start with

echo $CLASSPATH    <-- check setting of your Java class path
```

Note result: _____

_____

_____

```
echo $PATH              <-- check default places to look for
                            executables (such as the Java compiler!)
```

Note result: _____

_____

_____

Notes:

1. the pax command points to the directory where J510files.pax.Z was placed

2. Remember z/OS UNIX commands and file names are case-sensitive

**-- more --**

Now, from your J510 directory:

* compile Sample1.java
* run Sample1.class
* edit Sample1.java and add an output line with your name in it
* re-compile Sample1.java
* re-run Sample1.class

If your run fails, try adding your directory to the CLASSPATH list, for example:

**export CLASSPATH='/u/*userid*/J510:'$CLASSPATH**

If this solves the problem, you will have to ensure the CLASSPATH is set this way each time you run a lab. Consider setting up a shell script to simplify the process, or adding this line to your .profile script.

# Section Preview

- **Java Fundamentals**
  - **Data Types**
  - **Variables**
  - **Literals**
  - **Assignment Statement**
  - **Assignment Statements and Conversions**
  - **Assignment Operators**
  - **Other Operators**
  - **The Java Character Set**
  - **Unicode in Java**
  - **Identifiers**
  - **Java Operators (Machine Exercise)**

# Data Types

❒ **Java uses two types of data, Primitive types and Reference types**

- ♦ **Primitive types**
  - ✗ **boolean** - can have a value of true or false
  - ✗ Numeric - with these specific sub-types
    - ➢ **byte** - with values -128 to +127
    - ➢ **short** - with values -32768 to + 32767
    - ➢ **int** - with values -2147483648 to +2147483647
    - ➢ **long** - with values -9223372036854775808 to 9223372036854775807
    - ➢ **char** - with values '\u0000' to '\uffff', that is Unicode characters with decimal values from 0 to 65535; from version 5.0 on, Java supports surrogate pair characters also
    - ➢ **float** - as 32-bit IEEE single-precision floating point values
    - ➢ **double** - as 64-bit IEEE double-precision floating point
      - ➤ Note that floating point values include NaN (for not a number - the result of unallowed operations such as dividing by zero), negative infinity, negative numbers, signed zero, positive numbers, and positive infinity
      - ➤ Also, a Java implementation may optionally include float-extended-exponent and / or double-extended-exponent
      - ➤ Details of float data are beyond the scope of this course

# Data Types, 2

- **Reference types**
  - ✗ **class** type
  - ✗ **array** type
  - ✗ **interface** type

❒ **The programmer has direct access to variables of primitive type**

- **But variables of reference types provide only indirect access (think of a pointer to an entry in a table where you can't access the table yourself)**

# Data Types, 3

❒ **Every expression and every variable (field) is one of the above types**

- ♦ **Types limit / determine the values allowed for each expression and variable, and also determine the default value for variables**

- ♦ **Note that class types may be supplied or defined as part of the Java language (such as the String class) or user-supplied / defined (such as any user-created .class file)**

  - ✗ Similarly for interface types and array types, which we discuss later

- ♦ **There are also references in the docs to a "special null type" and to variables of type Object**

  - ✗ The first is to account for the null reference (indicating the reference variable is, effectively, empty or un-assigned)

  - ✗ The second is used to hold a null reference or a reference to any object, whether class, interface, or array type

❒ **Again, it's important to note that character string data ("String") is not a primitive data type**

- ♦ **The String data type is a class reference data type: String is a class, but a special class with several usability features built in**

  - ✗ We will discuss these in detail later; in the meantime pay attention to how we use String data in order to get used to working with it in a practical way

# Variables

- **A variable is a named storage location; it always is declared as one of the types described above**
  - **A variable of one of the primitive types always holds a value of that type**
  - **A variable of a class type may contain any of**
    - ✗ A null reference
    - ✗ A reference to an instance of the class
    - ✗ A reference to an instance of any subclass of the class
  - **A variable of array of some type may contain any of**
    - ✗ A null reference
    - ✗ A reference to an array of the type
    - ✗ A reference to an array of a reference type
  - **A variable of an interface type may contain any of**
    - ✗ A null reference
    - ✗ A reference to any instance of any class that implements the interface

# Variables, 2

- **Variables for reference types (classes, interfaces, and arrays) do not contain the address of the item, but a "reference" to the item**
  - **How the item is actually located and manipulated is hidden from the programmer**

- **In Java, a variable must be defined, physically in the code, before it may be referenced in an expression or statement**
  - **When a variable is defined, it does not need to have a value assigned because each variable will have a default value assigned, depending on the variable type**

  **Default values**

  - **Numeric types (byte, short, int, long, float, double) default to a value of zero**
  - **char type variables default to '\u0000'**
  - **boolean type variables default to** false
  - **reference type variables default to** null

- **However, locally declared variables do not take on a default value, and must have a value explicitly defined or assigned the first time the variable is used**

# Variables, 3

❒ **Variables are also grouped into one of seven kinds of variables**

- ♦ **Class variable - a variable shared by all objects in the class; declared with the** static **keyword in the class declaration**
- ♦ **Instance variable - a variable that has a unique location for each instance (object); does not have** static **specified**
- ♦ **Local variable - generally used in support of the logic of a method, constructor, or initializer, such as a loop control or counter variable; declared inside a block and only known inside that block**
- ♦ **Array components - unnamed variables created whenever a new array is created; referenced using subscript notation**
- ♦ **Method parameters - names specified in the declaration of a method identifying data passed to the method**
- ♦ **Constructor parameters - names specified in the declaration of a constructor, identifying data passed to the constructor**
- ♦ **Exception-handler parameters - names specified in the declaration of an exception handler**

# Literals

❒ **For each data type, you generally need to have a corresponding syntax for a literal of that type - for initialization, comparison, and so on:**

- ♦ **boolean - the reserved words** true **and** false
- ♦ **byte - decimal number in the range -128 to + 127**
- ♦ **short - decimal number, octal number (number '0' followed by base 8 digits), or hexadecimal number (number '0' followed by letter 'x' folllowed by hexadecimal digits) in range -32768 to +32767 (up to 4 hex digits)**
- ♦ **int - " w/ range -2147483648 to +2147483647 (up to 8 hex digits)**
- ♦ **long - " w/ range -9223372036854775808 to +9223372036854775807; followed by a letter L; (up to 16 hex digits)**
- ♦ **char - single quoted character, including Unicode escape characters (backslash followed by a letter "u" followed by four hex digits) and a few special escape sequences: \n - new line; \b - backspace; \t - tab; \f - form feed; \r - carriage return; \' - single quote; \" - double quote; \\ - backslash**
- ♦ **float - number with digits and decimal places; use E or e for scientific notation, followed by F or f**
- ♦ **double - number with digits and decimal places; use E or e for scientific notation; may follow with D or d to indicate 64-bit double (this is the default if not specified)**
- ♦ **String - double quoted strings, including Unicode escape characters**

# Literals, 2

- **Note that the reserved word "null" is valid for any reference type (but not primitive types)**

- **Note also that Unicode escape characters may be used anywhere in Java code, for example in names and labels, not just in literals**

- **Finally, there is a special kind of literal that applies to any class data type: specify the name of the class and append ".class"**

  - **For example: String.class refers to the object that represents the String type**

    - ✗ Which covers some subtleties we'll omit until we need to explore them

# Assignment Statement

❒ **The Java assignment statement is of the form**

*variable* **=** *expression* **;**

**Where**

- *variable* **is a Java variable as just discussed**
- *expression* **is one of these:**
  - ✗ A literal
  - ✗ Another variable
  - ✗ A method reference
  - ✗ A combination of literals, variables, and method references combined using various Java operators

*expression* **is evaluated and the result placed into** *variable*

- **There may be whitespace before and after the equals sign, or not**

# Assignment Statements and Conversions

❒ **Java is mostly strongly typed, which means, for example**

- ♦ **For assignment statements, the type of the variable has to match the type of the expression**
- ♦ **In general, you cannot have expressions that mix data types**

**However, there are a few exceptions**

- ♦ **Generally, things behave as you would like and expect**
- ♦ **You can mix numeric types as long as the type of the target can hold the largest possible value of any of the components of the expression**
- ♦ **You can explicitly cast a type, if it makes sense; for example if** amount **is declared as float and you assign it the value from** calcWork **which is defined as double, you can code:**

```
amount = (float) calcWork;
```

✗ and Java will convert the value in calcWork to float and assign the result into amount

➢ If the conversion is not possible (for example, the value in calcWork is too large to fit into a float variable), an exception is thrown (exceptions are discussed later)

# Assignment Operators

- **In addition to the classic "=" operator, Java supports the following variations of assignment (most of which are also found in C/C++ and some other languages):**

  - **+= add value on right side to variable on left side and place result into variable on the left side (qty += 5;)**

  - **-= subtract value on right side from variable on left side and place result into variable on the left side (amt -= 2.50;)**

  - ***= multiply value on right side by variable on left side and place result into variable on the left side (bal *= rate;)**

  - **/= divide value on right side into variable on left side and place result into variable on the left side (rslt /= scrs;)**

  - **%= divide value on right side by variable on left side and place remainder into variable on the left side (mod %= 7;)**

  - **&= bitwise AND value on right side to variable on left side and place result into variable on the left side (lc &= 0x42;)**

  - **^= bitwise XOR value on right side to variable on left side and place result into variable on the left side (uc ^= 0x20;)**

# Assignment Operators, 2

- **|=  bitwise OR value on right side to variable on left side and place result into variable on the left side  (workB |= 2;)**

- **<<=  shift value in variable on left side left the number of bits specified by value on the right side and place result into variable on the left side  (part <<= 3;)**

- **>>=  shift value in variable on left side right the number of bits specified by value, propagating the sign bit, on the right side and place result into variable on the left side  (part >>= 2;)**

- **>>>=  shift value in variable on left side right the number of bits specified by value on the right side, filling with binary zeros, and place result into variable on the left side  (rslt >>>= 3;)**

# Other Operators

❒ **Other operators, some of which we discuss later, may be used in expressions; here are a few:**

- **+ Concatenation of String literals and variables**
- **+, -, *, /, % Arithmetic operators based on previous page**
  - ✗ Also ++ and -- unary operators (increment and decrement)
    - ➢ Note: ++*variable* is different from *variable*++, in terms of when the variable is changed (likewise for --)
- **>>, <<, >>> Shift operators based on previous page**
  - ✗ Also ~ (flip bits), ! (not)
    - ➢ Note that ~ doesn't apply to boolean data and ! doesn't apply to numeric data
  - ✗ Also & (AND), ^ (exclusive OR), | (inclusive OR) - work as applied to integer (bitwise operations) or boolean (logical evaluation) types

**Codepage issues**

- **For z/OS, be sure ^ has a hex value of '5F', | has a hex value of '4F', and ! has a hex value of '5A'**

# Unicode in Java

- **The Java specification states that input elements to a Java program are formed using Unicode characters**
  - **In particular, UTF-16**
  - **However, most platforms use ASCII / UTF-8 for text files such as program code**
- **The Java compiler allows programs to be written using only ASCII characters**
  - **If you need a character not available in ASCII, say Φ, you can enter the character as a Unicode escape, which itself is composed solely of ASCII characters**
    - ✗ Unicode escapes are strings of the form \u*xxxx*, where *xxxx* represents a sequence of four hexadecimal digits
- **Of course, z/OS editors use EBCDIC (and then Java translates to UTF-16 as part of the compile process)**
  - **Unicode escapes work here, also**

# Unicode in Java, 2

❒ **A java source program, when compiled, is first processed to translate the source, including any Unicode escapes, into all Unicode; then the resulting string is broken into input elements:**

- **Whitespace - space, horizontal tab, form feed and line terminators (CR, LF, and CRLF)**
- **Comments - /* text */ - may cross line boundaries, and**
  **// text - may not cross line boundaries**
- **Tokens - five kinds:**
  - ✗ Identifiers - names and labels
  - ✗ Keywords - components of the language
  - ✗ Literals - raw data
  - ✗ Separators - nine choices: ( ) { } [ ] ; , .
  - ✗ Operators - discussed a few pages ago, with more later

❒ **Each kind of token has a role to play in the language and we will be discussing them all**

# Java Keywords

❒ **The following words are reserved as part of the Java language (so they should not be used as labels, names, or other identifiers):**

| | | | | |
|---|---|---|---|---|
| **abstract** | **continue** | **for** | **new** | **switch** |
| **assert** | **default** | **if** | **package** | **synchronized** |
| **boolean** | **do** | **goto** | **private** | **this** |
| **break** | **double** | **implements** | **protected** | **throw** |
| **byte** | **else** | **import** | **public** | **throws** |
| **case** | **enum** | **instanceof** | **return** | **transient** |
| **catch** | **extends** | **int** | **short** | **try** |
| **char** | **final** | **interface** | **static** | **void** |
| **class** | **finally** | **long** | **strictfp** | **volatile** |
| **const** | **float** | **native** | **super** | **while** |

**Notes**

- const **and** goto **are reserved even though they are currently not used**
- true **and** false **are not reserved words, they are boolean literals**
- null **is not a reserved word, it is the null literal**

# Identifiers

- **An *identifier* is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter**
  - **It must not be the same as a Java keyword, including the literals** true**, **false**, and** null
  - **Identifiers are used as variable names, code labels, enums, and so on**
- **Java letters include**
  - **uppercase and lowercase ASCII Latin characters (a-z, A-Z)**
    - ✗ Java is case sensitive
  - **The underscore (_) although it is included only for historical reasons**
  - **The dollar symbol ($) - although this is discouraged; intended for symbols created by mechanical code generators and for access to pre-existing names on legacy systems**
  - **Any character for which the Character.isJavaIdentifierStart(*int*) method returns** true **(this includes most Unicode characters except digits, whitespace, and control type characters)**
- **A Java letter-or-digit is any character for which the Character.isJavaIdentifierPart(*int*) method returns** true

# Identifiers, 2

❒ **Typical identifiers should seem reasonably obvious, and there will be lots of examples**

- **We have already seen...**
  - ✗ Sample1 - class name
  - ✗ total, counter, amount, calcWork - used as variable names
- **Java supports identifiers in any language**
  - ✗ However, for characters that are not EBCDIC you have to use Unicode escapes
  - ✗ For example:

```
char \u305f\u3061 = 'B';
```

➢ Instead of

```
char たち  = 'B';
```

**Note: "ta-chi" means "disposition", but this is just for illustration anyway**

Computer Exercise: Java Operators

Make a copy of Sample1.java, call it OpsTests.java.

Edit OpsTests.java and make these changes:

* change all Sample1 to OpsTests

* define three new variables:

```
byte oneByte = -63;
char oneChar = 0x41;
boolean oneBoolean = true;
```

* add lines to display the results of various operators being applied to total, oneByte, oneChar, and oneBoolean; for example:

```
System.out.println(" ");
System.out.println("total += 5 = " + (total += 5));
```

Include the following tests in the order specified (left to right, top to bottom):

| | | |
|---|---|---|
| total -= 6 | total *= 7 | total /= 8 |
| total %= 9 | total &= 5 | total ^6 |
| total \|= 7 | total <<= 3 | total >>= 2 |
| total >>>= 2 | | |

also, display before and after of ++total, total++, --total, total--

then include this final set of tests:

~total ~oneByte ~oneChar !oneBoolean

# Section Preview

- **Packaging**
  - **Classes**
  - **Inheritiance**
  - **Objects**
  - **.java Files**
  - **Nested Classes**
  - **Packages**

# Classes

❒ **A class is a named collection of**

- ♦ **Fields (attributes, variables, properties) that may apply to the entire class or to each object in the class**
  - ✗ Examples: name, description, quantity, unitPrice
- ♦ **Methods (routines, behaviors) that define the operations that may be performed on the class as a whole or on objects in the class**
  - ✗ Examples: setName, getName, verifyFormat

❒ **Examples of classes: BankCheckingAccount, Customer, Employee, CreditCardAccount, InventoryItem**

**Naming conventions (not required but recommended):**

- ♦ **Field names should be nouns (maybe with an adjective), mixed case: lowercase first letter, uppercase the first letter of each embedded word after the first**
- ♦ **Method names should be verbs or verb phrases, lowercase first letter and uppercase first letter of each word after the first**
- ♦ **Class names should be "descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized"**

# Inheritance

❒ **One of the main drivers of the object-oriented approach to computer systems is the ability to inherit characteristics from another class**

- ♦ **Thus, if one class, say BankAccount, has various fields (say customerName and accountBalance) and various methods (say accountDeposit and accountWithdrawal) ...**

- ♦ **... then any subclass or derived class (say, CheckingAccount) would be able to use the fields and methods of its parent class automatically: no new code need be written**

  - ✗ While preserving the option to have additional fields and methods or to implement a method differently if specified

❒ **There is one special class, named "Object", that is the root class, the top class, the starting class in the entire class hierarchy**

- ♦ **Technically it should be mentioned as "Object.class", but common usage is just "Object"**

- ♦ **Every class is a direct subclass of Object if no other parent class is specified**

  - ✗ Every class is descended from Object, directly or indirectly

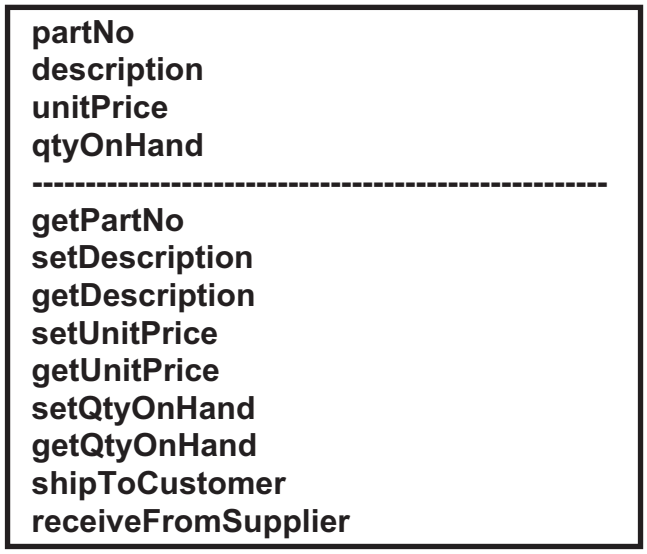
- ♦ **A class in Java may have only one parent class, but there may be a long line of ancestors ...**

# Inheritance, 2

❒ **Let's try a visual example ...**

❒ **First, suppose we represent a class in general as a rectangle containing fields and methods, and here's a specific example:**

**class InventoryItem**




♦ **this class itself is a child of class Object**

✗ Object is the parent class of InventoryItem

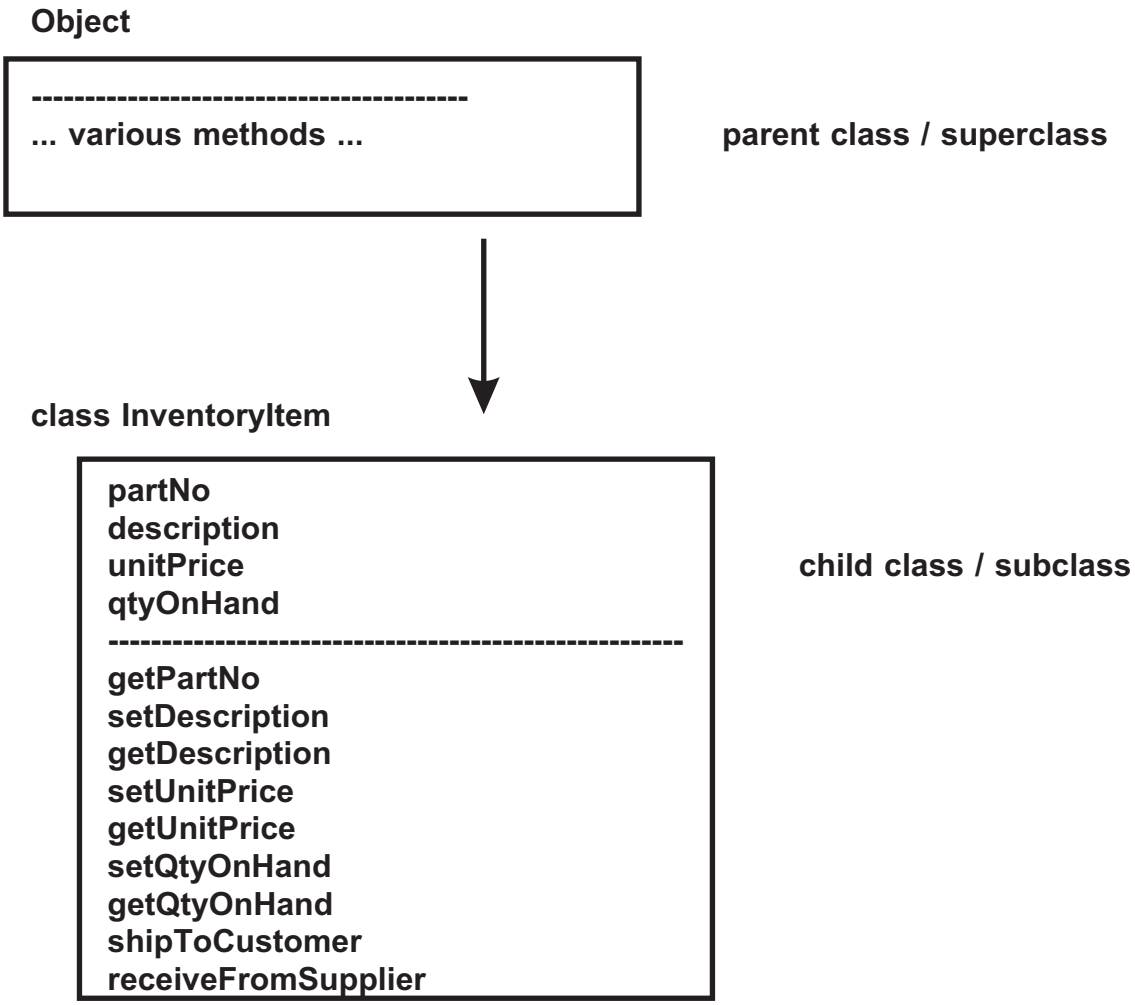
✗ InventoryItem is a **subclass** of Object

➢ We also say that InventoryItem **extends** Object

✗ Object is the **superclass** of InventoryItem

# Inheritance, 3

❒ **So we see the Object and InventoryItem classes in this relationship:**




♦ **In addition to its own fields and methods, InventoryItem inherits the fields (well, none for the Object class) and methods of Object (the methods of Object are a little esoteric for this point in the class)**

✗ So the child class is larger than the parent class; the subclass has more than the superclass - sort of counter-intuitive

# Inheritance, 4

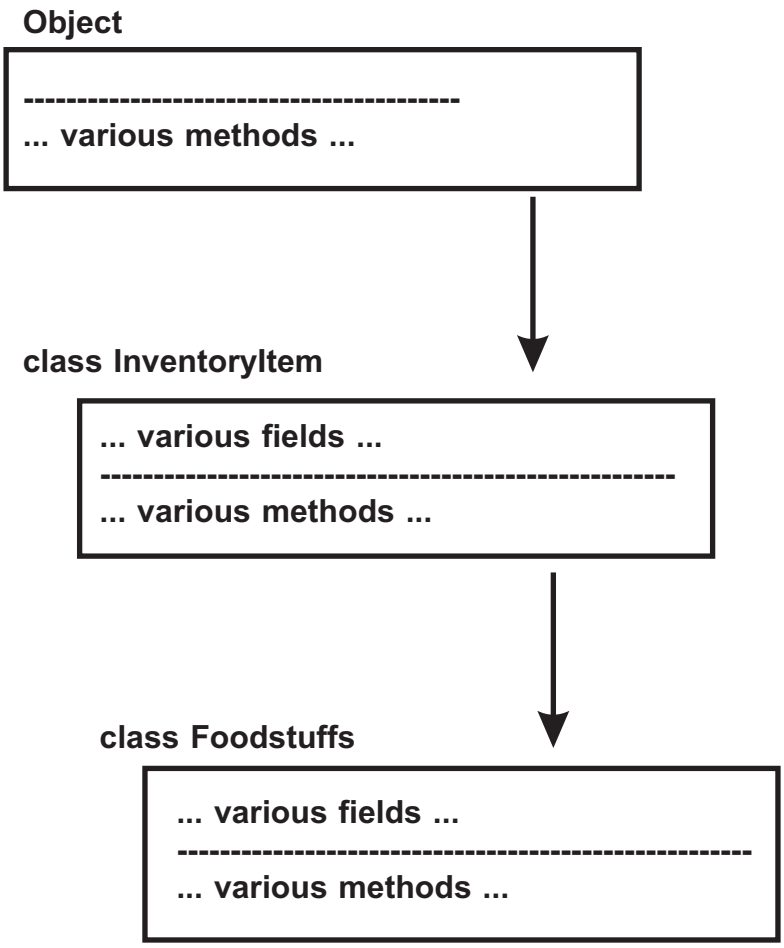
❒ **So, suppose the InventoryItem class allows us to process orders and shipments just fine**

- ♦ **But now we want to expand our product line: we want to carry some food items, say**
- ♦ **But many food items have an important attribute not carried in our InventoryItem class: an expiration date**

❒ **We have several alternative approaches we can take here**

- ♦ **Simply add the expirationDate field to our class, and modify and add to our existing methods**
  - ✗ This approach can disturb a lot of work already in place
- ♦ **Create a new class, Foodstuffs, as a child of InventoryItem**
  - ✗ As a subclass, Foodstuffs automatically has access to all the fields and methods of InventoryItem
    - ➢ Plus it can add fields and methods or override any of those in InventoryItem

# Inheritance, 5




## Notes

- **Every class has exactly one parent class, or superclass (except the Object class, which has none)**
- **A class may have any number of child classes (or subclasses)**
- **The fields and methods of a parent class are automatically available to its child classes**
  - ✗ However, this does not necessarily go back further then one generation, since at each generation a class can override fields and methods from its immediate parent

# Objects

❒ **An object is a dynamically created instance of a class or a dynamically created array**

- ♦ **When an object is created, we say it is instantiated - that is, an instance of the class shows itself as an object**

❒ **To work with an object, we must first instantiate it - create a variable of this type using the** new **operator, for example:**

```
Foodstuffs workItem = new Foodstuffs("Yellow pops");
```

- ♦ **Now you can display, add to a database, delete, update fields, and so on, using the workItem variable**
  - ✗ Lots of examples as we go on

❒ **The values of a reference type variable are references to objects**

- ♦ **That is, in the above example the content of workItem is a reference to a Foodstuffs object**
  - ✗ workItem is not the object, just a variable holding a reference to the object

# .java Files

❒ **Every .java file contains a complete definition of a class or an interface (we focus on classes at first; interfaces are covered later)**

- ♦ **That is, the file contains the definition of the class and of each field and method in the class and any variables needed to support the logic in the methods**

  - ✗ Note that a .java file may actually contain the definition of multiple classes or interfaces

❒ **Newer versions of Java include the additional constructs of enums and annotations - these may also appear in .java files**

- ♦ **Think of an enum as a special kind of class and an annotation as a special kind of interface**

  - ✗ Discussed later

❒ **In the Java documentation, the word type is often used as a shorthand for "classes, interfaces, enums, and annotations"**

# Nested Classes

❒ **A file that contains the definition of a class may also contain definitions of nested classes or enclosed classes**

- **Classes with their definitions totally included within the main class definition; structurally:**

```
public class Samp {   // Definition of class Samp
   .
   .
   .
   class Thimble {   // Definition of Thimble
      .
      .
      .
                  }

   static class Whistle {   // Definition of Whistle
      .
      .
      .
                           }

}
```

**Notes**

- **Nested classes are not subclasses**
- **A nested class declared "static" is called a "static nested class", and methods from these are accessed using the enclosing class name, *e.g.*: Samp.Whistle.getTune();**
- **A nested class that is not declared with "static" is called an inner class**

  ✗ An instance of an inner class can only exist within an instance of its containing class: a Thimble object can only exist within a Samp object

# Nested Classes, 2

❒ Inner classes are used primarily to implement helper classes

- ♦ For example, if you need to handle user-interface events, the event handling mechanism makes extensive use of inner classes

❒ There are two special kinds of inner classes

- ♦ Local inner classes - named inner classes defined within the body of a method
- ♦ Anonymous inner classes - unamed inner classes defined within the body of a method

❒ To put it all together, you can have...

```
class declaration {class definition}
class declaration {class definition
      Nested class(es)
         static nested class
         Inner class (non-static nested class)
               Local inner class
               Anonymous inner class
      }
```

❒ And, to make it explicit again:

- ♦ A class declaration is a header
- ♦ A class definition is the content after the header in the { } block

# Packages

- The Java Developers Kit and Java Runtime Environment come with thousands of classes and interfaces available for use

- To help organize things, classes and interfaces are grouped into packages - collections of related classes and interfaces

- Actually, every Java program is organized into packages implicitly, it's just that the default is to belong to an unnamed package

- Then packages are typically stored together in JAR files (Java ARchive files)

  - Which is basically a file in the classic ZIP file format, with an additional manifest that describes the contents of the JAR file

- Whether you are developing an application to sell on the software market or simply creating classes and applications for your organization, you need to pay attention to packaging issues right from the start

  - We'll use packages initially, then learn how to build and use JAR files later

# Packages, 2

**Problem: name collisions**

❒ **In addition to providing a way to organize thousands of pieces of code, packages provide a way to eliminate or minimize name conflicts**

**For example**

- **Suppose you create a class called "Receivables", with methods "locateInvoice", "checkInvoice", "updateAmount", "firstNotice", and "secondNotice"**

- **And another team in your department is working on a "Payables" class - they might also have a method named "updateAmount"**

  ✗ It has a different functionality, but that's a perfectly reasonable name for the application

- **Now, when the classes are brought together under the "BookKeeping" application any class that invokes "updateAmount" creates an amibguity to the runtime**

  ✗ Well, actually, it appears at compile time!

# Packages, 3

## Solution: packages

- **If you classify all your classes as belonging to the package named "incomes" and the other team uses a package name of "outgos" (note the lowercase first letter in both cases), then you can use these techniques when referencing the classes and methods from outside those packages**

  - ✗ **Fully qualify class references with package names, for example:**

```
incomes.Receivables
          workRcvbl = new incomes.Receivables;
```

  - ✗ **Then references to instance method names implicitly carry the package name; for example:**

```
totalDue = totalDue
        + workRcvbl.updateAmount(invNo);
```

  - ✗ **Alternatively, you could import specific classes into your application, using the import statement:**

```
import incomes.Receivables;
```

  - ✗ Then you can code:

```
Receivables workRcvbl = new Receivables;
```

**and**

```
totalDue = totalDue
        + workRcvbl.updateAmount(invNo);
```

# Packages, 4

**Solution: packages, continued**

✗ **Import all classes and interfaces from your package:**

```
import incomes.*;
```

✗ **Then you can use the same technique for other classes as we demonstrated for the Receivables class**

❒ **However, note that if you import all packages in the application:**

```
import incomes.*;
import outgos.*;
```

✗ Then you are back to the situation where ambiguous class and method names must be fully qualified

✗ However, at least there will be no conflicts with classes and methods from other sources

# Packages, 5

❒ **A package consists of any number of compilation units, each of which has any of these parts (each is optional, but there must be at least one of them present; if present, they must appear in this order):**

- **A package declaration, giving the fully qualified name of the package to which the compilation unit belongs**

  ✗ A compilation unit that has no package declaration is part of an unnamed package

- **Zero or more import declarations that allows types from other packages and static members of types to be referred to using their simple names**

- **Top level declarations of class types and interface types**

  ✗ Below these, for class types, is the definition of each class in terms of its attributes, fields, and methods

- **All these pieces are called members of the package**

❒ **Each compilation unit automatically has access to all types declared in the same package**

  ✗ And also automatically imports all the public types in the predefined package **java.lang**

    ➢ The literature uses the term "types" to mean what we know as primitive types but also classes and interfaces

# Packages, 6

❒ **Package names have a complex set of guidelines / rules, designed to minimize name collisions among many classes and methods:**

- **All packages supplied with Java have names that begin with java. or javax.**
- **It is suggested that companies could start their package names with their website URL reversed**
  - ✗ For example: at The Trainer's Friend, our URL is **trainersfriend.com**, so we should use **com.trainersfriend** as the high level name qualifiers
  - ✗ Deeper levels of names might reflect departments and projects, and so on
- **It's really a short sighted guideline: suppose a company gets bought out or renamed, or just changes its URL?**
  - ✗ The essential point to remember is the objective: keep names unambiguous
- **Package names should be short, descriptive, and all lowercase**
- **Package names may contain Unicode characters not in the displayable or keyable set by using Unicode break characters; for example:**

**children.activities.crafts.papierMâché**

**coulde be written as:**

**children.activities.crafts.papierM\u00e2ch\u00e9**

# Packages, 7

❒ **The** import **declaration identifies sources for classes (packages)**

**Syntax**

**import** *package_name*.*member_name*;
**- import a particular member**

**import** *package_name*.*;
**- import all members in a package**

**Rules**

- **A class may have multiple** import **declarations**
- **These must come at the front of the .java file**
  - ✗ There may be comments before, and there may be a package declaration before, but nothing else
- **Package names do not form a hierarchy; for example, there are packages named java.awt, java.awt.color, and java.awt.font - but these last two are not subsets of the first; they are independent packages (actually they are subpackages)**

**Note**

- **An** import **statement is not like a "copy" or "include" statement in other languages: there is no source nor object physically copied into the code**
  - ✗ **import** just identifies where to locate classes and interfaces at compile time and run time

# Packages, 8

❒ **But we have to pay attention: how do you think the Java compiler finds packages and their classes?**

- ♦ **The Java Language Specification says it is implementation dependent**
  - ✗ But for z/OS UNIX (and, in fact, for most [if not all] UNIX and Linux platforms), the implementation uses the native directory structure

❒ **Suppose your current working directory is J510, and this is the directory where you will build your application code**

- ♦ **Planning ahead, you see a need for some packages**
  - ✗ At least, say, one package that will handle our inventory items, providing classes and their related constructors, fields, and methods
- ♦ **So, create a subdirectory of J510, call it inventory**
- ♦ **And create all inventory related classes in this subdirectory**
- ♦ **Each .java file in the inventory subdirectory will start with a package declaration: package inventory;**
- ♦ **Each .java file in the J510 directory that needs these packages will start with an import declaration:**
  **import inventory.*;**

# Packages, 9

❒ **Now suppose you need to create some classes that would be useful in both the classes in J510/inventory and other classes elsewhere in the application**

- ♦ **You could create a subdirecotry of inventory, called, say, helper**
- ♦ **So you now have J510, J510/inventory, and J510/inventory/helper directories**
  - ✗ Note that for Windows platforms, you would use the backslash, but the same hierarchy
    - ➢ So **J510\inventory** and **J510\inventory\helper** and so on

# Packages, 10

❒ **In general, then, to create packages for your own applications**

- **Place a package declaration at the front of each .java file that is considered to belong to the package**

**Syntax**

**package** *package_name*;

- **These .java files all belong in a subdirectory with the same name as** *package_name*
- **Note that any .java file may import packages**
- **Note also that the term** subpackage **is only defined in the Java docs indirectly, through statements such as these:**
  - ✗ "Each host determines how packages, compilation units, and subpackages are created and stored"
  - ✗ "an unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package"
  - ✗ "The package java has subpackages awt, applet, io, lang, net, and util, but no compilation units"

# Packages, 11

❒ **Also note that the directory hierarchy for your applications and your packages needs to be included in your CLASSPATH environment variable**

- **It is sufficient to include the top directory in the hierarchy, since the compiler knows how to drill down from there**
- **For example, on my home system I have a home directory of /u/scomsto**
- **If I build /u/scomsto/J510 and /u/scomsto/J510/inventory and /u/scomsto/J510/inventory/helper, then if /u/scomsto/J510 is in my CLASSPATH, then...**
  - ✗ Java files in **helper** need to start with

```
package inventory.helper;
```

  - ✗ Java files in **inventory** need to start with

```
package inventory;
import inventory.helper.*;
```

  - ✗ Java files in **J510** that will be using any of these packages need to begin with:

```
import inventory.*;
import inventory.helper.*;
```

# Packages, 12

❒ **If /u/scomsto/J510 is not in your CLASSPATH, but, say '/u/scomsto' is, then all of the package names above would have to begin with J510 (on both package declarations and import declarations)**

- **In other words, your CLASSPATH has to include from root down to the directory where your application is positioned, so that subdirectories get found properly**

  ✗ Package names need to begin with the first subdirectory below that

- **This allows you to use relative paths, which gives you some flexibility in deploying your applications and packages**

- **Note you separate the levels in a package name with periods so that Java can make the translation depending on the platform**

  ✗ The periods become forward slashes on UNIX-like systems and backward slashes on Windows systems

- **You must use this naming convention also for running a class inside a package directory, that is:**

  **java** *package_name*.*class_name*

❒ **Note that if you are providing your Java apps as a product, you might want to put your .class files in one set of directories**

- **And your .java files in another (in order to protect your source)**

- **This may require attention vis a vis packages and your directory structure**

# Packages, 13

❒ **We shall use packages in all our labs and lectures that use our case study - in order to get used to using this construct**

- ♦ **We shall start in the next section, with the example we've been discussing, so that the Java code that defines inventory related classes will be coded and compiled in a subdirectory named "inventory"**

- ♦ **This may also require you to set your CLASSPATH correctly each time you get into z/OS UNIX**

  - ✗ We provide a script you can modify so that when you will be doing Java work you can run this script to set up the CLASSPATH

    - ➢ Named **setJavaPackagePath**

❒ **All classes in a package are automatically available to each other, as are their methods, fields, and constructors (discussed soon)**

- ♦ **Except that methods, fields, or constructors designated as** private **or** protected **are not visible**

❒ **To be available to classes outside of their package, however, classes, constructors, methods and fields need to be designated as** public **(and those classes outside the package still need to have the appropriate** import **declarations, of course)**

❒ **A class may be** public **or not; a constructor, method, or field may have zero or one of** public**,** protected**,** private**, or** static **specified - these terms are called access modifiers and are discussed in more detail soon**

# Section Preview

- Case Study Part 1
  - The Story
  - InventoryItem Class - First Cut
  - InvRept Class - First Cut
  - Lessons From Our First Cuts
  - Compiler Options
  - Starting An Application (Machine Exercise)

# The Story

❒ **From here on out, we will examine Java using a particular, concrete (but somewhat simple) example**

- ♦ **Introducing terms, concepts, syntax only as we need it to solve the next step of our development**
- ♦ **Trying to always go with the minimal approach to get the job at hand done**

❒ **Our application is this:**

- ♦ **We have a business that warehouses, sells, and ships a variety of products, mostly novelties and gag gifts - most sales are done online via our website**
- ♦ **We want to create a Java application that accomplishes the following:**
  - ✗ **Scan the transaction log for the past 24 hours and create a report that lists all items sold**
  - ✗ **Scan the inventory database and the transaction log togehter and list all the items that did <u>not</u> sell in the past 24 hours**
    - ➢ And this report should put out a grand total of the current value of our inventory
  - ✗ **Both reports are to be put out in XHTML format on our corporate intranet, available to management for viewing and analysis**

# The Story, 2

❒ **Whenever you design a Java application, you must ask the questions:**

- ♦ **What are the classes and objects the application deals with? A first look might yield this list:**
  - ✗ Items in inventory
  - ✗ Items in the transaction log
  - ✗ XHTML report files (headers, details, trailers)
- ♦ **What functions need to be performed on these classes and objects? Clearly, some of these**
  - ✗ Accessing data in the files, including looping and detecting end of file
  - ✗ Comparisons against some key field
  - ✗ Calculations for grand total
  - ✗ Construction of XHTML files

❒ **To gain some practice and experience, we want to just define a class that describes items in inventory in a minimalist way, so ...**

# InventoryItem Class - First Cut

- Just to get a simple start, we define our class this way, in the file InventoryItem.java:

```
package inventory;
// Declaration of class InventoryItem
  public class InventoryItem {

     String PartNo;
     String Description;


    // Definition of method getPno
        public String getPno()
        {
          return PartNo ;
        }


    // Definition of method getDesc
        public String getDesc()
        {
          return Description ;
        }


 }   // end of class definition InventoryItem
```

- Here we have two fields and two methods that retrieve the value in those fields

  - Note that the class and its methods have the "public" access modifier, so they may be referenced by classes outside thier package

# InventoryItem Class - First Cut, 2

❒ **Notes on the class:**

- **We start with a package declaration, indicating which package this class belongs to (so it should reside in the inventory subdirectory)**
- **Next, a comment (not required, but not a bad idea)**
- **Then we declare the class with "public" and the name of the class**
  - ✗ Notice the opening brace ("{") - this begins the definition of what is meant by or included in the class (notice also there is no semicolon here)
  - ✗ And the end of the definition is the close brace ("}") at the very end
  - ✗ Notice: declaration versus definition
- **Next we declare the fields / attributes / data items using the syntax of: data_type data_name semicolon**
- **Then we declare and define the two methods using the construct:**

**access modifier of** public
**type of data returned (***e.g.***: "String")**
**method name (***e.g.***: "getPno")**
**followed by parameter list ("()" - none here)**
**open brace ("{")**
**definition of body: what does the method do?**
**(in this case, simply "return" a value)**
**close brace ("}") - terminates definition of method**

# InventoryItem Class - First Cut, 3

❒ **Oh, wait: how do we create one of these items?**

- ♦ **We need to add a constructor**

❒ **A constructor is a special block of code that defines how the fields in an object are initialized when the application creates a new instance of the object (instantiates the class)**

- ♦ **Every class must have at least one constructor**
  - ✗ If you do not code one, a default constructor is supplied that takes no arguments and that sets fields to their default values
- ♦ **A class may have multiple constructors (details shortly)**
- ♦ **All constructors must appear before methods in the class definition**
- ♦ **All constructors must have the same name as the class, for example:**

```
InventoryItem()
```

- ♦ **Finally, constructors, like methods, should be declared "public" so they may be accessed by other classes that import the package this class is found in:**

```
public InventoryItem()
```

# InventoryItem Class - First Cut, 4

- Typically a constructor is passed an argument list that determines or influences the values used to initialize the new object, for example:

```
public InventoryItem(String pno, String desc)
```

  - This says: if an application creates a new InventoryItem object and passes two strings, use the code in this constructor method

- You might have, for example, another constructor method to use when no data is passed - a method to build an object with default values; its header might look like:

```
public InventoryItem()
```

  - This constructor would override the default no-argument constructor

- Let's look at our class with two such constructor methods ...

# InventoryItem Class - First Cut, 5

```
package inventory;
// Declaration of class InventoryItem
   public class InventoryItem {

      String PartNo;
      String Description;

      // Definition of constructor methods

         public InventoryItem()
          {
             PartNo = "Part0000";
             Description = " ";
          }

         public InventoryItem(String pno, String desc)
          {
             PartNo = pno;
             Description = desc;
          }


     // Definition of method getPno
        public String getPno()
          {
             return PartNo ;
          }


     // Definition of method getDesc
         public String getDesc()
         {
            return Description ;
         }


  }   // end of class definition InventoryItem
```

# InventoryItem Class - First Cut, 6

- **So if a Java application encounters a request to build a new object, how does the class know which constructor code to use?**
  - **By the arguments passed when the "new" attribute is specified**
    - ✗ If no arguments are passed, an InventoryItem object is created with a PartNo of "Part0000" and a Description of a single space
    - ✗ If two string arguments are passed, an InventoryItem object is created with a PartNo that contains the value of the first argument and a Description that contains the value of the second arguemnt
    - ✗ If any other argument mix is passed, it is an error (or "exception")
  - **This is called the signature of a constructor (or method): the name combined with the types of parameters it expects**
    - ✗ So what arguments are passed to a constructor or method implies which version of the method is used
    - ✗ This is called overloading (sometimes "function overloading")

**Note: you pass arguments but receive parameters**

- **Its the same data, just different points of view**

# InventoryItem Class - First Cut, 7

❒ **Just one more note about constructors:**

- ♦ **Constructors may do more than initialize fields**
  - ✗ Could prepare files or databases
  - ✗ Could issue messages
  - ✗ Could clean up from other work
  - ✗ ...

# InventoryItem Class - First Cut, 8

- **Something we have lost track of: classes and objects are descriptive - we need some code to use these items**
  - **We need a class that drives some kind of application**
  - **Some ideas for applications:**
    - ✗ Get field values and build a data store
    - ✗ Generate an HTML page of inventory status
    - ✗ Take orders and generate invoices
    - ✗ Lots more ...
  - **We'll head for building an HTML inventory status page, but we have to begin simple, so we create a class called InvRept that will display values ...**

# InvRept Class - First Cut

```
import inventory.*;
// Declaration of class InvRept
  class InvRept {   /* Definition of class InvRept
                              - a Java application   */

  // Declaration of method "main"
  public static void main(String[] args) {

  // Definition of "main"
       System.out.println("In main");
       InventoryItem TestItem =
             new InventoryItem("PART00305", "Hogwash");
       System.out.println("Partno = " +
             TestItem.getPno());
       System.out.println("Description = " +
             TestItem.getDesc());
       System.out.println("Leaving main");

  } // end of method "main"

 }  // end of class definition InvRept
```

**Notes**

- **Notice the** import **declaration, showing were to find references to classes in packages**
- **Notice the comment: /* ... */ on lines 3 and 4**
  - ✗ This style comment may cross any number of lines
- **Recall that each Java statement is terminated by a semicolon**
- **Blocks are signified as all lines between an opening brace ("{") and its corresponding close brace ("}")**
  - ✗ Blocks themselves are not terminated by a semicolon: just the statements inside the blocks

# InvRept Class - First Cut, 2

**Notes, continued**

- **A Java application always includes a single method named "main" - and it is this method that is invoked automatically when you run the top Java class in the application**

- **Notice the declaration of main - we will discuss all the pieces shortly**

- **Notice also the syntax for instantiating an object:**

  *class_name* *object_reference_name* **=**
  **new** *class_name*(*args*);

  ✗ When such a line is executed, a new object reference item is created and the appropriate constructor method is implicitly invoked to initialize the object; from the example:

```
InventoryItem TestItem =
        new InventoryItem("PART00305", "Hogwash");
```

- **Also note the syntax used to invoke methods against this object:**

  *object_reference_name*.*method_name*(*args*)

  ✗ From the example:

```
...  TestItem.getPno()  ...
```

- **Finally note that if the result of invoking a method is a value of some kind, the invocation needs to be treated syntactically as a noun (an operand in a compare, say, or in an arithmetic or string expression); but if a method returns** void **(no value), the invocation of the method should be coded as a verb (***e.g.* **System.out.println() )**

# InvRept Class - First Cut, 3

❒ **Now, given these two classes in their .java files and in the appropriate directories and subdirectories, the way to proceed is:**

- ♦ **javac InventoryItem.java** — **- compile "subroutine" (from inventory subdirectory)**
- ♦ **javac InvRept.java** — **- compile "main line" (from base directory)**
- ♦ **java InvRept** — **- run main routine (from base directory)**

❒ **And the results we see are:**

```
In main
Partno = PART00305
Description = Hogwash
Leaving main
```

❒ **OK. Now let's explore this a little more**

- ♦ **How is it that the class "InventoryItem" is located from the separately compiled InvRept?**
- ♦ **Can we combine these files into one?**

# InvRept Class - First Cut, 4

❒ **When you compile a .java file, by default the resulting .class file is put into the same directory as the .java file**

- ♦ **The compiler ("javac") is found using environment variable PATH**
- ♦ **The Java Virtual Machine (JVM, invoked by "java") is found using the same variable**
  - ✗ The JVM is loaded and the specified class file is found in one of the directories pointed at in the CLASSPATH environment variable
    - ➢ Usually this list includes ".", which means, the current working directory
  - ✗ When the running application invokes a method against a class, the CLASSPATH directories are also searched until the class is found (if not found, the application terminates with an error)
    - ➢ Of course, package directories are also searched, if there is an **import** declaration for one or more packages

❒ **Note: if you compile a .java file that invokes a class in a different file in the same directory, if that .java file has not been compiled, javac will locate and compile the source for the invoked class as well as the named .java file**

- ♦ **javac will also re-compile invoked classes from .java files in the same directory if the .java file has been changed and the changed version has not yet been compiled**

# InvRept Class - First Cut, 5

- As far as combining the .java files into a single file, experiment shows that this works:

```
// Declaration of class InvRept
 class InvRept {
  // Definition of method "main"
  public static void main(String[] args) {

  // Logic of "main"
       System.out.println("In main");
       InventoryItem TestItem =
             new InventoryItem("PART00305", "Hogwash");
       System.out.println("Partno = " + TestItem.getPno());
       System.out.println("Description = " + TestItem.getDesc());
       System.out.println("Leaving main");
  } // end of method "main"

 }   // end of class definition InvRept
```

```
// Declaration of class InventoryItem
        class InventoryItem {

    String PartNo;
    String Description;

    // Definition of constructor method
      InventoryItem(String pno, String desc)
       {
          PartNo = pno;
          Description = desc;
       }

   // Definition of method getPno
      String getPno()
       {
         return PartNo ;
       }

   // Definition of method getDesc
      String getDesc()
       {
         return Description ;
       }

 }   // end of class definition InventoryItem
```

# InvRept Class - First Cut, 6

**Notes**

- **Here we simply copied one file after the other (note: no packages, no imports, only "main" has to be public)**
- **We also tried putting the InventoryItem definition totally inside the InvRept body, but moving closing brace to the end, like:**

```
// Declaration of class InvRept
 class InvRept {
  // Definition of method "main"
  public static void main(String[] args) {
.
.
.
  } // end of method "main"


// Declaration of class InventoryItem
        class InventoryItem {
.
.
.
 }   // end of class definition InventoryItem

 }   // end of class definition InvRept
```

- **But this didn't work**
- **But if we change the declaration of InventoryItem to be static:**

```
static class InventoryItem {
```

✗ Then it compiles clean - producing two class files: **BigInvRept$InventoryItem.class** - a nested class, and **BigInvRept.class**

✗ And a run: **java BigInvRept** is successful as is: **java BigInvRept$InventoryItem**

# Lessons From Our First Cuts

❒ **So, to summarize the salient points of our case study so far:**

- **Classes are declared with a header that describes:**
  - ✗ Zero or more class modifiers
    - ➢ So far we have only seen "public" and "static"
      - ➤ "public" basically means the class is visible everywhere; "static" is only allowed for nested classes
  - ✗ The reserved word "class"
  - ✗ The name of the class folllowed by an opening brace ("{")
- **Then the defintion of the class, which may include fields and methods**
- **A closing brace ("}")**

- **Fields are declared as a data type (*e.g.*: "String" or "int", *etc.*) followed by the name of the field and, optionally, an initial value**

- **Methods are declared with a header that describes:**
  - ✗ Zero or more method modifiers
    - ➢ So far we have seen two: "public" and "static"
      - ➤ "public" and "static" must both be specified for the "main" method, and may be specified elsewhere, which will be discussed when the need arises
  - ✗ The type of data the method returns when invoked ("void" if no data are returned); always a single item
  - ✗ The name of the method followed by a set of parenthesized comma-separated descriptions of the arguments the method expects (In the style of fields described above) followed by an opening brace
- **Then the definition of the method followed by a closing brace**

# Lessons From Our First Cuts, 2

❒ **Furthermore, a class must have one or more constructors**

- ♦ **Constructors look like methods, but the name of the constructor must be the same as the name of the class**
- ♦ **Constructors are not considered to be methods (so they are not automatically inherited by subclasses, discussed later)**
- ♦ **Each constructor has its own signature, consisting of the class name and the type of arguments it expects**
- ♦ **Note: there may be methods with duplicate names; methods have signatures the same as constructors: the name of the method and the type(s) of arguments they receive, as mentioned on page 73**
  - ✗ It's important to note that a class may not have two methods with the same signature and different return types

❒ **Multiple classes may be declared in a single .java file - sometimes called a compilation unit**

- ♦ **Although the common practice is to generally define separate classes each in their own file, so they may be used by many other classes**
  - ✗ More on this later

# Lessons From Our First Cuts, 3

❒ **A Java application has at its top level a class with a "main" method**

- ♦ **This method always takes a single argument, which is descirbed as an array of character strings and returns void**
  - ✗ Recall:

```
public static void main(String[] args)
```

- ♦ **We discuss how to use the** args **data soon**

❒ **We also have seen how packages fit into the role of an application**

- ♦ **Structurally, create a subdirectory to hold all the .java and / or .class files that are related**
  - ✗ Each must have a "package" declaration at the front
  - ✗ Note that all classes in a package are considered "friends" - they can access each others' methods without needing to import each other
- ♦ **Applications that use packages must have import declarations for each package they wish to access**
  - ✗ Except for classes in the same package
- ♦ **The head directory of your packages must appear in your CLASSPATH environment variable**

# Compiler Options

❒ **The javac command has a number of options you may specify when you compile a .java file**

♦ **Specify as:** **javac [-*options*] [*file(s)_to_compile*]**

**Most useful options at this point**

| | |
|---|---|
| **-verbose** | **output messages during compile** |
| **-classpath** *path*<br>**-cp** *path* | **path to look for user classes; overrides any value in the CLASSPATH environment variable** |
| **-sourcepath** *path* | **path to look for file(s) being compiled** |
| **-d** *directory* | **directory where to place generated class files** |
| **-deprecation** | **show a description of each use or override of deprecated members or classes (usually such members have been supplanted with newer features, and the older version may be removed sometime in the future)** |
| **-encoding** *codepage* | **specify the codepage the source is in** |
| **-help** | **display complete list of javac options** |

❒ **So, for example, you could invoke a compile this way:**

```
javac -classpath /u/scomsto -d /u/stnt329/Java/inventory Inventory.java
```

## Computer Exercise: Starting An Application

In your J510 directory you should have the files setJavaPackagePath, InventoryItem.java and InvRept.java as discussed in lecture. Start building our application as follows:

1. Modify setJavaPackagePath to reflect your directory structure. Copy this script to your home directory. Run the script as:

   . setJavaPackagePath

   (remember the leading period; you will want to run this script once each time you come to work on course labs)

2. Make a subdirectory, inventory, and move InventoryItem.java into it

3. Compile InventoryItem.java from J510/inventory and InvRept.java from J510 (in that order)

4. Run InvRept and ensure the output is what you expect.

5. Modify InventoryItem.java as follows:

   * Add two new fields:

     An integer named QOH (for Quantity On Hand)

     A floating point number named unitPrice

   * Add two additional parameters to the constructor header, so it now looks like this:

     InventoryItem(String pno, String desc, int qty, float unprice)

   * In this constructor add two assignment statements to accept the new values into the appropriate field names

   * Add two new methods, getQOH and getUnPr to provide ways to access the current values.

6. Compile InventoryItem.java. If you are curious, try the -verbose flag to see what happens

Computer Exercise, p.2.

7. Modify InvRept.java as follows:

   * The line where the code creates a new InventoryItem object, add two additional values to the argument list: 5 and 12.50f

     (the 5, of course, is the quantity on hand, and the 12.50f is the unit price; the letter 'f' is present because if you just code a literal 12.50 Java treats this as a double instead of a float value, and the assignment will fail)

   * Add two lines to display the results of invoking getQOH and getUnPr

8. Compile InvRept.java;

9. when you get a clean compile, run InvRept.

This page intentionally left almost blank.

# Section Preview

- **Command line arguments, Arrays, and Control of flow**
  - **Command Line Arguments**
  - **Arrays**
  - **The Java if Statement**
  - **Java Conditional Operators**
  - **Java Control of Flow Statements - switch and break**
  - **Java Control of Flow Statements - for**
  - **Java Control of Flow Statements - while and do while**
  - **Java Control of Flow Statements - break, continue, return**
  - **Command Line Arguments and Arrays (Machine Exercise)**

# Command Line Arguments

- Recall that all "main" methods (at least the ones we've seen so far) begin with this line:

```
public static void main(String[] args) {
```

- The arguments inside the parentheses provide a mechanism for you to accept data from the command line when the class is run
  - In particular, you can accept an array of Strings, and the name to use for the array is "args"

# Arrays

❒ **Arrays are their own kind of reference data type**

- **Define an array by a data type followed by brackets ("[ ]") followed by the name of the array: String[] args**
  - ✗ Note: you may also define an array by a data type followed by the name of the array followed by brackets: **float prices[]**, although this is discouraged
- **Arrays may have multiple dimensions: specify one set of brackets for each dimension: float[][][] lengthWidthHeight**
- **Elements in an array are referenced by specifying a subscript, as a literal or integer variable: args[2]**
  - ✗ The first element is subscript number zero
  - ✗ Of course you can code a loop that cycles thorugh all or some of the elements in an array - as soon as we talk about loops!
- **The number of elements in an array is known as the length of the array, and you can access that by** *array_variable_name***.length**
  - ✗ Note that the length of an array is not part of its type, which is why you declare an array with no value in the brackets

# Arrays, 2

- **You can initialize single elements by assignment statements**

- **You can initialize multiple elements using an assignment statement with multiple values, separated by commas, enclosed in braces, at the time you define the array, for example:**

```
String[] lastName = {"Akers", "Bradbury"};
```

- **You can assign similar arrays to each other:**

```
int[]  arr1 = {1, 3, 5};
int[]  arr2 = {2, 4, 6};

arr1 = arr2;
```

✗ Now arr1[0] has 2, arr1[1] has 4, and arr1[2] has 6

❒ **You can set the size of an array by allocating memory using 'new':**

```
char[] inChars;
inChars = new char[100];
```

❒ **Note that, unlike C, a String is not an array of characters, and an array of characters is not a String**

- **And neither a String nor an array of characters are terminated by a null**

# Command Line Arguments, resumed

❒ **When a Java application is run from the command line, you may specify a string of [blank-delimited] words, for example:**

```
java GenPage first quarter summary
```

- **The number of words will be found in args.length (=3 in the example)**

- **Each individual word is accessible through args[*n*]**
  **in the example, args[0] is first, args[1] is quarter, and args[2] is summary**

- **If you do not pass any command line argument, args.length is zero - BUT any reference to args[0] throws an exception**

  ✗ An ArrayIndexOutOfBoundsException

- **So if we will be accessing the command line values, we'd better first check args.length for being zero**

  ✗ Which leads us to the **if** statement

  ✗ Which leads us to flow of control statements

  ✗ Which we'll follow by a discussion of how to read the Java docs

  ✗ And we really do need to talk more about String

**We have a full agenda!**

# The Java if Statement

❒ **The classic, premier, conditional statement in any language is simply "if", and Java is no exception**

**Syntax**

**if (***expression***)**
**{** *statement-block* **}**

**[else**
**{** *statement-block* **} ]**

**Notes**

- **The** if **statement does not end with a semi-colon, but any statements inside either** *statement-block* **do**
- *expression* **is evaluated and if the result is** true**, the first** *statement-block* **is run**
- **If** *expression* **evaluates to** false**, if there is an** else **clause, the second** *statement-block* **is run**
  - ✗ If there is no **else**, contol simply passes to the end of the **if** statement
- **The statements inside either** *statement-block* **may include almost any Java statement, including** if**; in other words,** if **statements can be nested**

# The Java if Statement, 2

**Examples**

```
   if (args.length > 0)
     {
System.out.println("args.length = " + args.length);
System.out.println("args[0] = " + args[0]);
     }
```

```
   if (pno == null)
    {
     System.out.println("Part number not supplied.");
     System.out.println("Using default value.");
     PartNo = "Part0001";
    }
   else
    {
     PartNo = pno;
    }
```

- **Notice the difference betwen comparing for equality (==) and the assignment statment (=), as with many languages**

  ✗ Also note we are comparing a String object to the null reference, a technique that will be important later

- **Of course, the expressions may be complex, for example**

```
if (((QOH + QOO) <= ReordQ )&& (D > 15)) {...}
```

# Java Conditional Operators

❒ **For any Java statement that uses a conditional test (we've just seen** if **so far, but there are others), conditional expressions involve one or more of these operators:**

| | |
|---|---|
| **==** | **Equal to** |
| **!=** | **Not equal to** |
| **>** | **Greater than** |
| **>=** | **Greater than or equal to** |
| **<** | **Less than** |
| **<=** | **Less than or equal to** |

- **Just one per test, of course, and such tests always result in a value of true or false**

❒ **To combine tests to build more complex conditional tests, use these operators:**

| | |
|---|---|
| **&** | **And** |
| **&&** | **And; only evaluate right hand expression if left hand expression is true** |
| **\|** | **Inclusive Or** |
| **\|\|** | **Inclusive Or; only evaluate right hand expression if left hand expression is true** |
| **>** | **Exclusive Or** |

❒ **You can also use this construct from C:**

*variable* **=** *condition* **?** *value1* **:** value2;

- **Which means if** *condition* **is true, assign** *value1* **to** *variable*, **otherwise assign** *value2*

# Java Conditional Operators, 2

❒ **One other test that's a little unusual, the** instanceof **operator, which returns** true **if an object is an instance of a class**

**Syntax**

*object* **instanceof** *class*

**Example**

```
if (WorkItem instanceof PerishableInventoryItem)
   { ... }
```

- **This could be a way to introduce subclasses and see which subclass an item belongs to without having to define some kind of "type" field or code to carry around**

- **Or a way to work with a general Object type variable that might hold a reference to any object**

# Java Control of Flow Statements - switch and break

❒ **The Java** swtich **statement is pretty much like the C version of the same statement: use it to implement the case structure**

**Syntax**

**switch***(field*)
**{**
**case** *value*$_1$**:** *statement(s)*; **break;**
.
.
.
**case** *value*$_n$**:** *statement(s)*; **break;**
**default:** *statement(s)*; **break;**
**}**

**Notes**

♦ *field* **must be one of these types: byte, char, short, int (that is, simple numerics / integers)**

✗ Or enumerated types (enums) and the wrapper classes Character, Byte, Short, and Integer - all of which we discuss later

♦ **The various** *value***s must be literals or enums**

♦ **The "break" statements are optional, but without them, once a match is found on a value, all subsequent statements will be executed; think of "break" as the way to terminate each separate** case**, since Java has no "goto"**

# Java Control of Flow Statements - switch and break, 2

## Example

```
char choice;
.
.
.
switch (choice)
  {
   case 'q': OrderItem.chgQty(inQty); break;
   case 'd': OrderItem.chgDesc(inDesc); break;
   case 'u': OrderItem.upUnPr(inUPr); break;
   default:  System.out.println("Unknown code");
  }
```

## Notes

- **Presumably the user has been prompted for a transaction code and an update value**
  - ✗ Presumably, also, "OrderItem" is an instance of InventoryItem, which has had some additional methods defined
- **Although the trailing "break" is a good idea in general, it is not needed with the "default" phrase, since that is always the end of the switch statement anyway**

# Java Control of Flow Statements - for

❒ **The** for **statement allows you to iterate some code over a range of values - usually an array or collection**

- **But it could be, for example, the months that have passed since a deposit was made or a loan was made, to calculate interest due**

**Syntax**

```
for (initialization; termination; iteration)
    {
        statement(s);
    }
```

**Where**

- *initialization* **takes one of these forms:**
  *type name*=*value* **if variable** *name* **not yet defined**
  *name*=*value* **if** *name* **already defined**

- *termination* **identifies the condition when the loop should end (if the condition is false, exit)**

- *iteration* **defines the rule for stepping through the items (usually indicates how incrementing should take place)**

- *statement(s)* **represents the block of Java statements to execute each time the for block is run**

# Java Control of Flow Statements - for, 2

**Example**

```
for (int i=0; i<args.length; i++)
  {
   System.out.println("args[" + i + "] = " + args[i];
  }
```

- **If compiled and invoked as:**

  **java myClass with these words**

- **... would display these lines:**

```
args[0] = with
args[1] = these
args[2] = words
```

❒ **All three operands of the** for **statement are optional! For example, if you coded:**

```
for ( ; ; ) {

   // logic in here executed indefinitely

}
```

- **Here is an infinite loop that requires a** break **or** return **statement to exit the loop (discussed shortly)**

# Java Control of Flow Statements - enhanced for

❒ **A new form of** for **is available on the latest versions of Java**

- **This kind of** for **statement, often called "enhanced** for**", is very useful for processing all the elements of an array or collection**

  ✗ Specify a list of elements to process - not even necessarily numeric elements!

**Syntax**

```
for (data_type variable_name : array_or_collection_name)
    {
        statement(s);
    }
```

- **The block of statments is executed while the value in** *variable_name* **ranges over all the values in the array or collection**

# Java Control of Flow Statements - enhanced for, 2

**Example**

```
int[] numbers - {1,3,7,12};
String[] Operations = {"add item", "display item"};

for (String items : Operations)
   { System.out.println("Operation = " + items);}

for (int item : numbers)
   { System.out.println("Current value is: " + item);}
```

- **Which when compiled and run yields:**

```
Operation = add item
Operation = display item
Current value is: 1
Current value is: 3
Current value is: 7
Current value is: 12
```

❒ **Very nice to have this kind of flexibility!**

# Java Control of Flow Statements - while and do while

❒ **These two statements support looping through a block of statements as long as some expression is true**

- ♦ **With the expression being tested at the top of the loop (while) or at the bottom (do while) - which is pretty evident from the syntax:**

**Syntax - while**

```
while (expression) {
    statement(s)
    }
```

**Syntax - do while**

```
do {
    statement(s)
    } while (expression);
```

- ♦ **In both cases,** *expression* **must be of boolean type**
  - ✗ That is, resolve to **true** or **false**

# Java Control of Flow Statements - while and do while, 2

## Example

```
String[] choices = {"add", "subtract", "done"};
boolean work_to_do = true;
int i;

    while (work_to_do)
     {
      for (i = 0; i<choices.length; i++)
       {
        if (choices[i].equals("done"))
         {work_to_do = false;}
        System.out.println("current choice is: " +
               choices[i]);
       }
     }
```

- **Which really just demonstrates syntax**

  ✗ **while** and **do while** are much more useful when processing entries in a table or array or until the user requests "quit" and that sort of thing

  ➢ Which we will cover in this course

## Special note

- **When you must compare a String with a literal or compare two Strings, you need to use equals(*string*), or a similar method; these are discussed later**

# **Java Control of Flow Statements -** break**,** continue**, and** return

❒ **There are three statements that let us break out of the flow in conditional or looping statements**

- ♦ **break - which we have already seen, but which we will expand on**
  - ✗ Can break out of **for, while**, or **do while** in addition to **switch**
  - ✗ Can use labeled break to break out to an outer loop
- ♦ **continue - in a loop skips the current iteration and goes to the loop control test**
- ♦ **return - leaves the current method, returning to where the method was invoked**

# Java Control of Flow Statements - break revisited

- **Earlier we saw how to use** break **to leave a** switch **construct once there has been a match (**case**)**

- break **may also be used to exit from a** for **loop, a** while **loop, or a** do **while loop**

  - **Sort of a limited-use, cheap "go to"**

- **The basic form of** break **leaves the innermost loop, if there are nested constructs (if there is no nesting, it just leaves the current loop)**

**Examples**

```
for (int i=0; i<args.length; i++)
  {
    if (args[i].equals("quit") break; // leave 'for'
   System.out.println("args[" + i + "] = " + args[i];
  }
```

```
String[] choices = {"add", "subtract", "done"};
boolean work_to_do = true;
int i;
    while (work_to_do)
     {
      for (i = 0; i<choices.length; i++)
        {
         if (choices[i].equals("done"))
          {work_to_do = false;
           break;}  // leave 'for' and 'while'
         System.out.println("current choice is: " +
                choices[i]);
        }
     }
```

# Java Control of Flow Statements - break revisited, 2

❒ **An additional feature of** break **is the ability to break out a containing loop that is labeled, either the current loop or a an outer loop containing the current loop**

- **A label in a Java program is an identifier (see pages 38-39) followed by a colon (:)**
- **This provides a name for a location in the code**
- **Cannot be used for jumping, since Java does not have a goto kind of statement**
- **But can be used to name a block of code, and to exit a block of code by name**

✗ We can revise the second example on the previous page as:

```
String[] choices = {"add", "subtract", "done"};
boolean work_to_do = true;
int i;
looper:
    while (work_to_do)
     {
      for (i = 0; i<choices.length; i++)
       {
        if (choices[i].equals("done"))
           {break looper;} // leave 'for' and 'while'
        System.out.println("current choice is: " +
                choices[i]);
       }
     }
```

# Java Control of Flow Statements - continue

❒ **This statement lets you jump out of the current iteration of a** for**,** while**, or** do while

- ♦ **Essentially a jump to the loop control logic from within a loop**
- ♦ **For example**

```
for (int i=0; i<args.length; i++)
  {
   if (args[i].equals("almost"))
      {continue}
   System.out.println("args[" + i + "] = " + args[i];
  }
```

✗ The **continue** here will increment **i** and then go to the test for **i<args.length**

# Java Control of Flow Statements - continue, 2

❒ **Like** break, continue **may specify a label, for example:**

```
out:  while(more.equals("yes"))
   {.
    .
    switch(Reply)
     {case '1': looking="yes"; break;
      case '2': more="no"; break out;
      default: System.out.println("Invaid value");
     }
    getpart: while (looking.equals("yes"))
     {.
      .
      if (src.compareTo("in")==0)
       {showing="no";
        looking="no";
        more="no";
        continue out; }
      for (int i = 0; i<items.length; i++)
       { if (scnd.compareTo(items[i])==0)
          showing="yes";
          showdata: while (showing.equals("yes"))
          {.
           .
           switch (Option)
            { case 'a': .
                        .
                        continue showdata;
              case 'b': more="yes"; showing="no";
                        continue getpart;
              case 'c': more="no"; looking="no:;
                        showing="no"; continue out;
              default: System.out.println("NO!");
                        continue showdata; }
          } // end of showdata
       } // end of if scnd==
      } // end of for (int i = 0
     } // end of getpart
    } // end of out
```

# Java Control of Flow Statements - return

❒ **The** return **statement is used to exit a method**

- ♦ **Control returns to the point after where the method was invoked**
  - ✗ We have seen examples already on pages 68, 72, and 80
- ♦ **The main method in an application does not need a** return

❒ **If the method is to return a value, the value is specified on the** return **statement itself, for example:**

```
return amount;
```

**or**

```
return (qty*price);
```

- ♦ **The result of evaluating the expression must be of the type expected to be returned**

❒ **If the method is not supposed to return a value (it is specified as void), then its** return **statement(s) must not have a value:**

```
return;
```

- ♦ **A method may have multiple return points; the first one encountered will cause the method to exit**

## Computer Exercise: Command Line Arguments and Arrays

In your J510 directory you have the file **Parser.java**. This file is a starting point to explore some of the themes discussed in this chapter. This is a stand alone class, so it needs no **import** nor **package** declarations.

The file contains:

* The usual header for a class including a main with args

* Declares for two integer arrays (called "odds" and "evens") and an array of characters (called "requests")

* An enter message and an exit message

You are to add the following pieces of code, between the enter and exit messages:

1. Display the number of words passed in from the command line

2. Display each of the words on a separate line
   (Hint: use a **for ( )** construct
   As you display the words, if any of them have the string "Java", display an extra message identifying which word that is (you could have multiple occurrences)

3. Display each of the values in the odds array
   (Hint: use a **for ( )** construct)
   As you display the values, also display the value + 5 and if one of the values is 3, display an extra message identifying which entry that is

4. Run through all the entries in odds (Hint: use a **for ()** construct) and for each one, **set the int named j to 0** and then use a **while** construct like **while((odds[i] + evens[j]) < 9))** to:
   * Display the current odds entry
   * Display the current evens entry
   * Display the sum of these values
   * Increment j

***** more *****

5. Run through all the entries in requests (Hint: use a **for ()** construct) and use a **switch ()** construct to display a message for the current entry:

| | |
|---|---|
| for 'a', display | **requests[i] is add** |
| for 'c', display | **requests[i] is change** |
| for all else dispay | **requests[i] is done** |

6. Compile Parser until you get a clean compile (Tip: take these parts one at a time, don't try to do it all at once.).

7. Test Parser; run it with no command line arguments, run it with one or more words on the command line; include the word Java at least once on the command line.

**Important note: The results may not be what you expect. If you get a clean compile but the results are surprising, see if you can figure out why the results are not what you expected. Don't stew over any surprises too long. Later we'll clarify it all.**

**Exercise stretch**: if you have time and interest, add some code using labels and labeled **break** and **continue** statements.

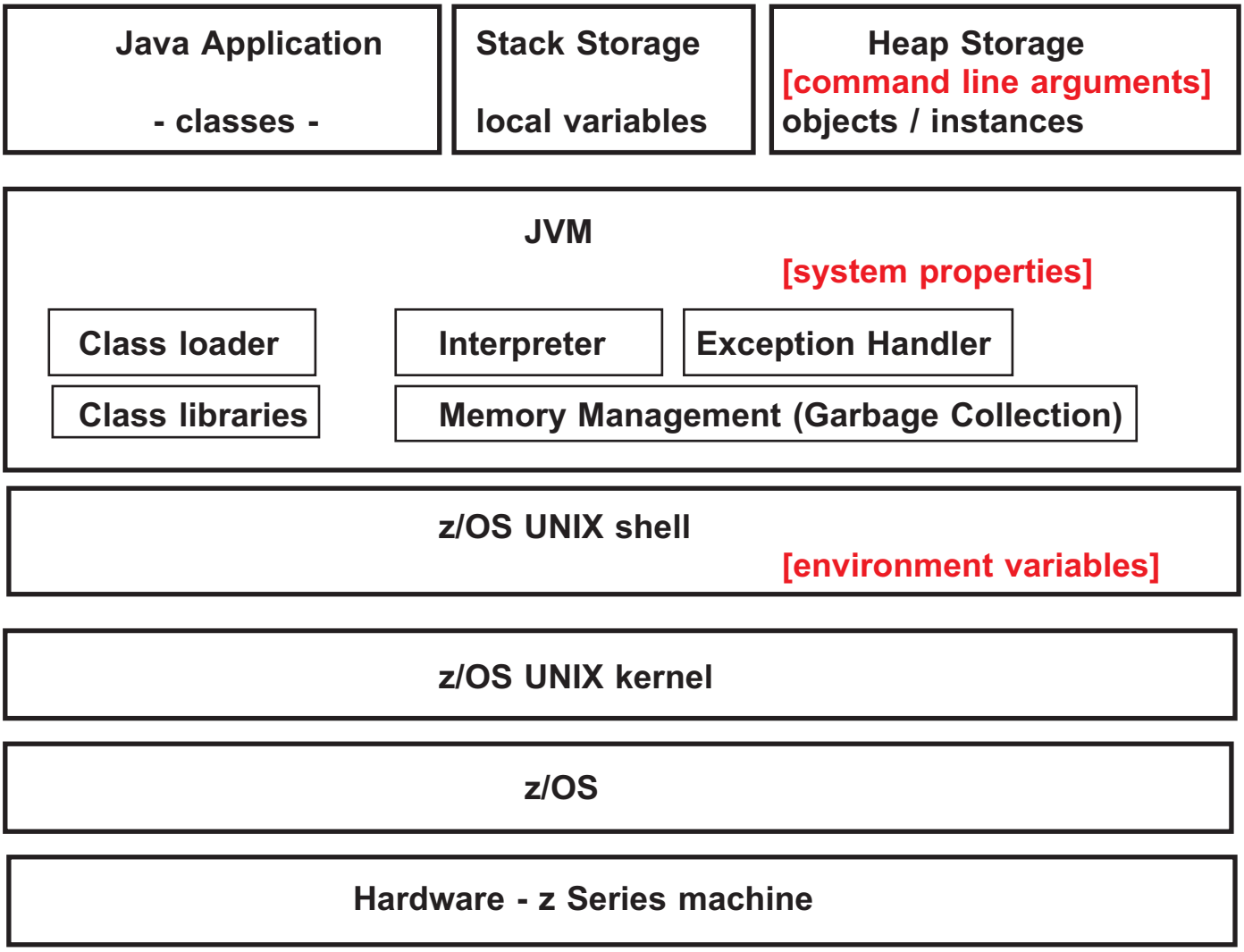
This page intentionally left almost blank.

# Section Preview

- **Java Docs, Part 1**
  - **The Java Runtime Environment**
  - **Java Docs - URLs**
  - **Java Docs - The Java Language Specification**
  - **Java Docs - JDK 6 Documentation**
  - **The java Command**
  - **System Properties**
  - **Java Docs and System Properties (Machine Exercise)**

# The Java Runtime Environment

- When you launch a java application, the java command brings up a Java Virtual Machine (JVM)
  - Which is software that runs java bytecode from a class
- The environment looks something like this, conceptually




- Note that command line arguments are set as part of the java command, system properties are defaulted but some may be set / defined on the java command line, and environment variables are set using UNIX commands (command line or script)

# The Java Runtime Environment, 2

- **Which brings up the question: how do we find out the options for the** java **command (or any part of Java?)**
  - **In this section, we will point you to a few starting points for reference about Java**
  - **And we discuss more later**

# Java Docs - URLs

- The content of this course is drawn from the official documentation about Java on the Sun Microsystems website
  - Along with some other books, discussions with colleagues, and help from various listserv's and newsgroups
- The content of this course is current as of the pubilcation date of the materials
- But ultimately you will find that you need to find the official documentation yourself, due to one or more of these reasons:
  - You need to explore some concept, syntax, facility in more detail than we covered in this course
  - Time has gone by and there are new versions / releases of Java

# Java Docs - URLs, 2

❒ **So, we suggest, first, you visit our web site where we have a free technical publication called "Reading Java Docs": http://www.trainersfriend.com/Papers/Reading_Java_Docs.pdf**

- **This 20-page paper describes how to set up the Java documentation from Sun on various servers to make them available**

  ✗ But it begins with a description of how to locate the central docs on the Sun site, the highlights of which we repeat here:

**The starting point**

**http://java.sun.com**

**The Java Language Specification**

**http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html**

**The JDK Documentation**

**http://java.sun.com/javase/6/docs/**

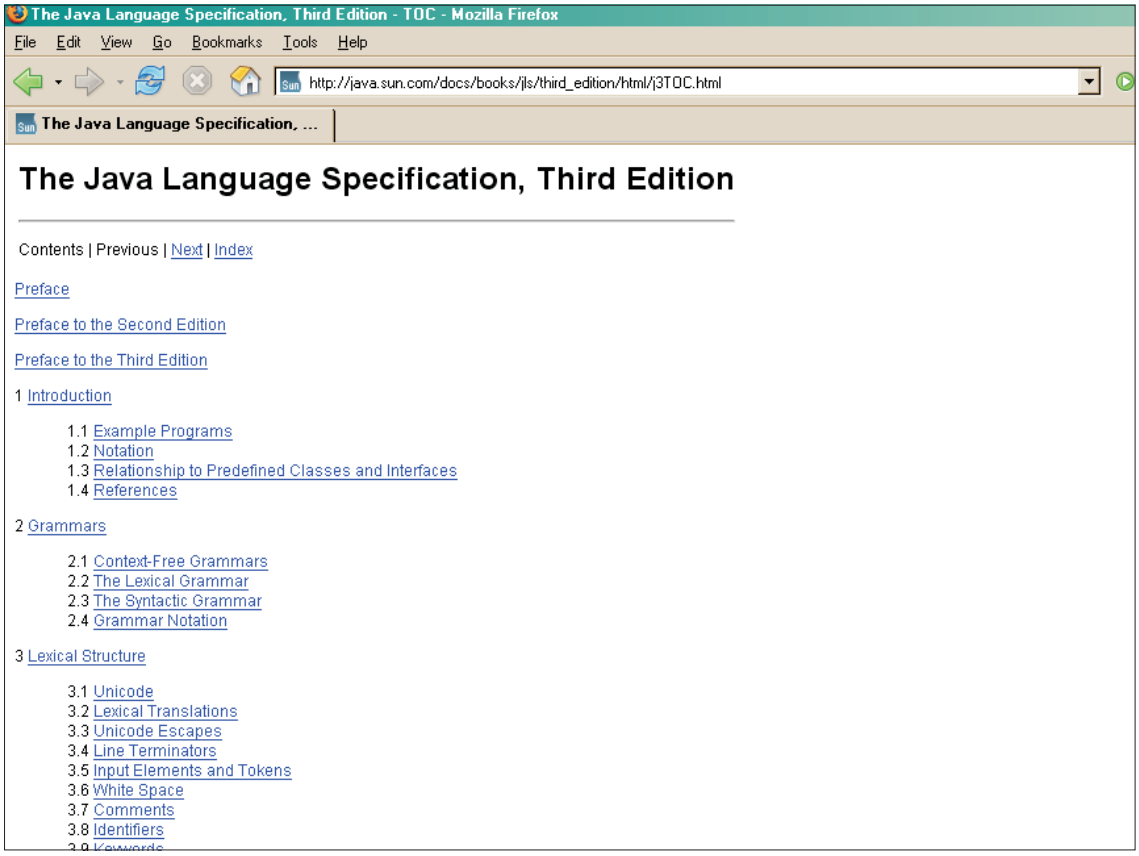
**The API / packages and classes descriptions**

**http://java.sun.com/javase/6/docs/api/**

❒ **We discuss the Java Language Specification and JDK Documentation pages in this section, and discuss the API's information in a later section**

# Java Docs - The Java Language Specification

❒ **If you follow the link to http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, you see this page:**



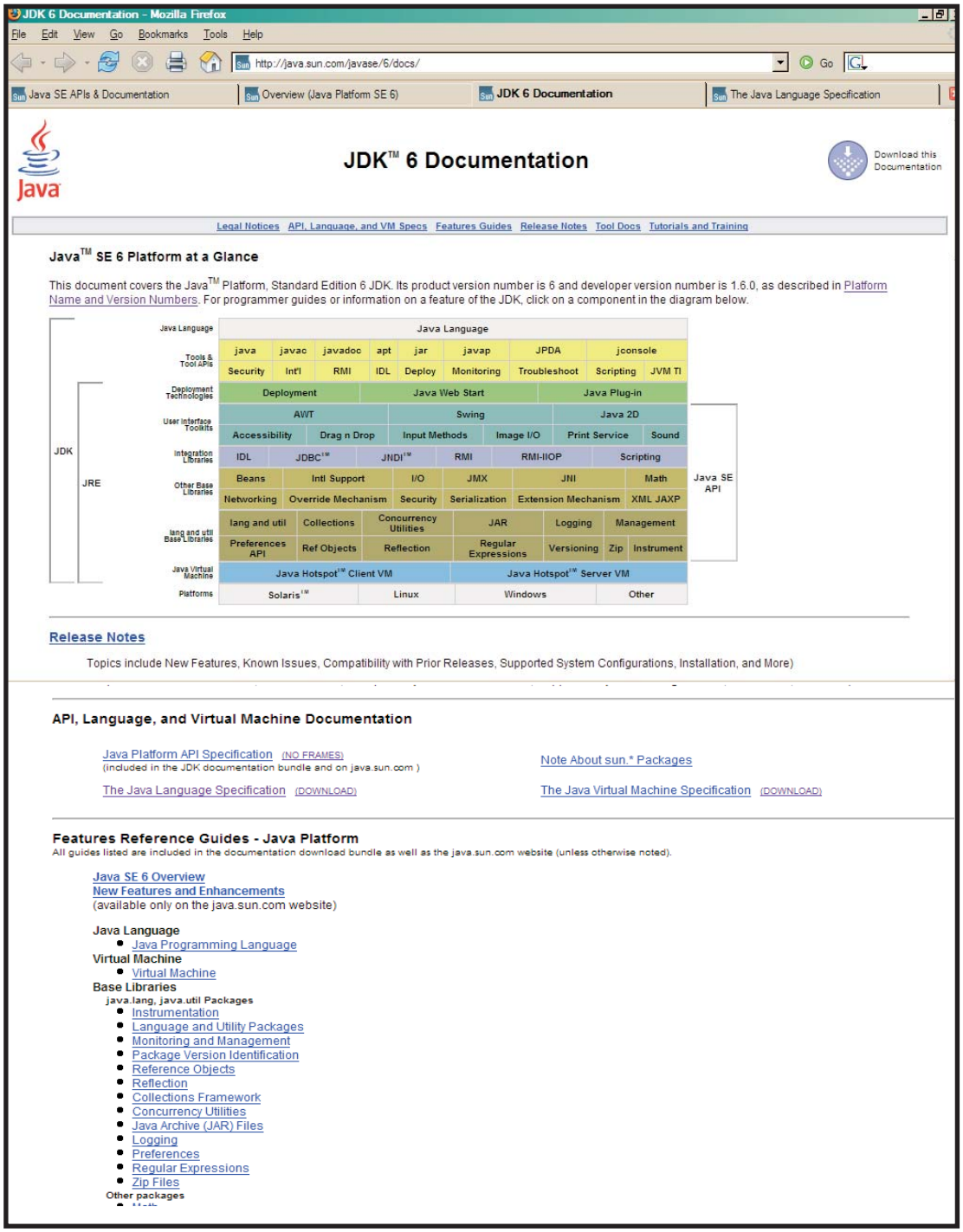

- **This page contains the table of contents for the precise specfication of the Java language**

  ✗ Dry, complex, arcane - but complete and definitive

- **Each numbered link takes you to a particular page, and the sub-numbered links (*e.g.*: 2.1, 2.3, ...) takes you to a particular anchor on that page**

# Java Docs - The JDK Documentation

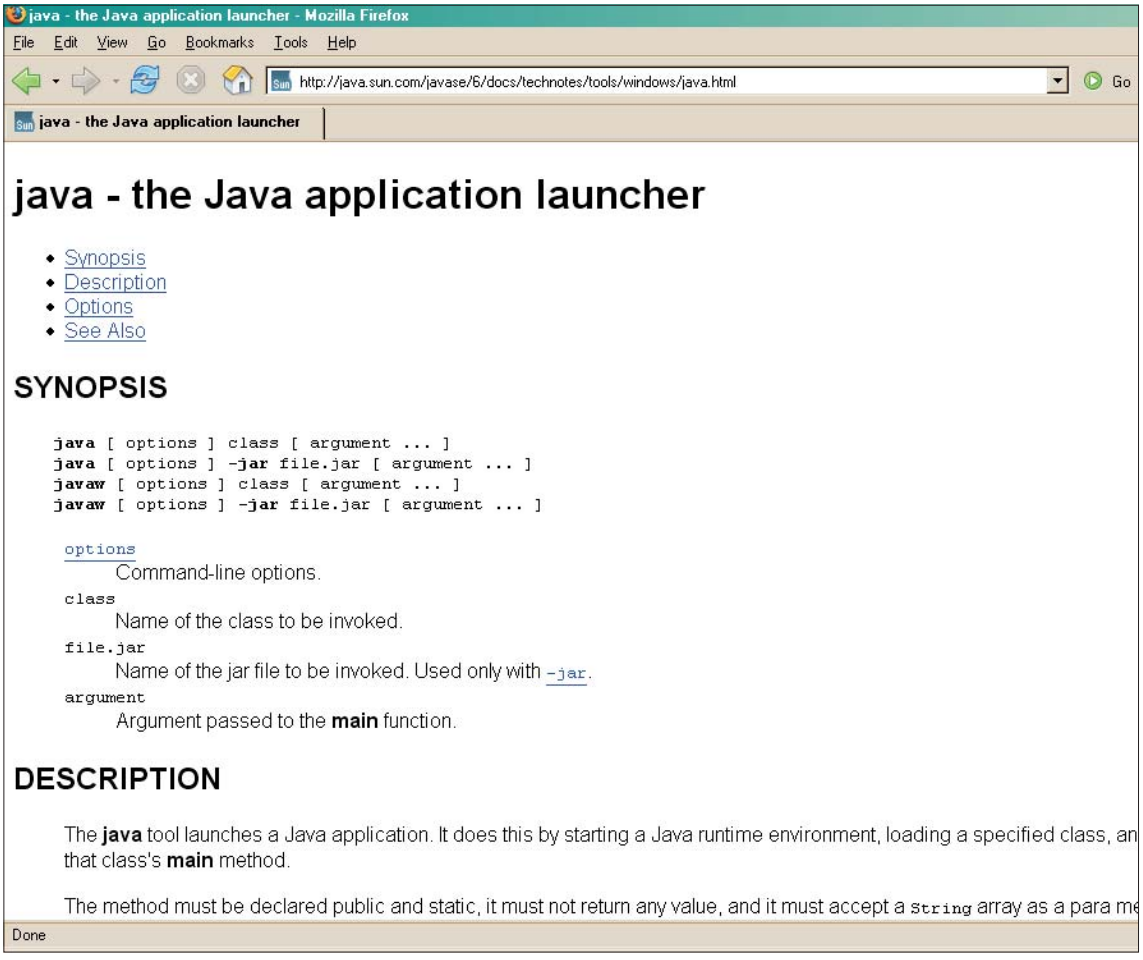
❒ **The link http://java.sun.com/javase/6/docs/ takes you to this page:**




♦ **We've shown two scrolls worth of this page here**

# The java Command

❒ **Each of the colored boxes is actually a link to the related information; following the link to "java" gets you here:**




- **And here you can find all you need / want to know about the java command**

# The java Command, 2

**Notes on the java command**

- **You can run with java or javaw; javaw runs the java code without a command window**
  - ✗ On z/OS, these are the same; might be some efficiency to run javaw instead of java
- **There are quite a number of options you could specify**
  - ✗ And we examine some on the next page
- **You can run a program found in a JAR file (we mentioned these on page 52)**
  - ✗ Which we'll discuss later
- **And you can pass a string of words as arguments**
  - ✗ We discussed both how to pass them and how to access these strings in the previous section

❒ **So now, let's examine the options you can specify on the** java **and** javaw **commands ...**

# The java Command

❒ **The interesting options on the java command, from the perspective or our current studies, are ...**

- **-cp or -classpath: you can specify a list of directories to search to locate classes while running this application**

  ✗ Note: if you specify this, it overrides any CLASSPATH value

- **-verbose: display information about each class loaded (might use for debugging or simply if you're curious)**

- **-showversion: display Java version information and then run the application**

- **-D*property*=*value*: set a system property**

  ✗ The system properties are both useful and helpful, so we cover them now

  ➢ The source for this information is found in a later section

❒ **There are a number of properties that you can access that describe the current JVM**

- **And some that you can set**

- **And you can create your own "system properties" simply through coding the -D option on the java command**

# System Properties

❒ **The following properties are always defined:**

| | |
|---|---|
| java.version | Java Runtime Environment version |
| java.vendor | Java Runtime Environment vendor |
| java.vendor.url | Java vendor URL |
| java.home | Java installation directory |
| java.vm.specification.version | Java Virtual Machine specification version |
| java.vm.specification.vendor | Java Virtual Machine specification vendor |
| java.vm.specification.name | Java Virtual Machine specification name |
| java.vm.version | Java Virtual Machine implementation version |
| java.vm.vendor | Java Virtual Machine implementation vendor |
| java.vm.name | Java Virtual Machine implementation name |
| java.specification.version | Java Runtime Environment specification version |
| java.specification.vendor | Java Runtime Environment specification vendor |
| java.specification.name | Java Runtime Environment specification name |
| java.class.version | Java class format version number |
| java.class.path | Java class path |
| java.library.path | List of paths to search when loading libraries |
| java.io.tmpdir | Default temp file path |
| java.compiler | Name of JIT compiler to use |
| java.ext.dirs | Path of extension directory or directories |
| os.name | Operating system name |
| os.arch | Operating system architecture |
| os.version | Operating system version |
| file.separator | File separator ("/" on UNIX) |
| path.separator | Path separator (":" on UNIX) |
| line.separator | Line separator ("\n" on UNIX) |
| user.name | User's account name |
| user.home | User's home directory |
| user.dir | User's current working directory |

# System Properties, 2

- A component of the JVM called the Security Manager can be set so you are not allowed to retrieve any of these values, but usually it's not a problem

- As mentioned, you can create your own system properties simply by making up a name in a -D option when you issue the java or javaw commands

  - This might be useful as an alternative to command line arguments

- You can access the value in a system property by invoking the System.getProperty() method:

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

Computer Exercise: Java Docs and System Properties

1. Take a few minutes and explore the Java docs sites discussed in this section.

2. Modify your Parser.java program: near the end, add some displays of these system properties:

   java.version
   java.vendor
   java.vendor.url
   os.name
   os.arch
   user.name
   user.home
   user.dir
   user.var

3. Compile Parser and test it; note that user.var is not a standard system property, so you will have to pass a value when you test, something like this:

   java -Duser.var="Part0015" Parser some words

❒ **Note: if you want to download the API docs to your workstation, go to: http://java.sun.com/javase/downloads/index.jsp and scroll down the page looking for Java SE 6 Documentation**

- ♦ **Installing is simple: download and unzip**

This page intentionally left almost blank.

# Section Preview

- **Case Study Part 2**
  - **The htmlGen class**
  - **Emitting HTML from Java (Machine Exercise)**

# The htmlGen Class

- **This is not an HTML course, it is a Java course**

- **But our case study involves emitting HTML to a file, which will then be viewed in a browser**
  - **So we assume you know a little HTML**
  - **And we have provided a class, htmlGen, that will emit html for you**
  - **Now seems like a good time to add this to our case study work**

- **The htmlGen.java file is provided for you, and it has these class level methods:**
  - **putStart() - takes no argument and emits five lines to compose the top of an HTML page**
  - **putCol(*string_value*) - takes a single string value and emits the value embedded in a <pre> element**
  - **putEnd() - takes no argument and emits </body> and </html>**

# The htmlGen Class, 2

- **To use this class, your InvRept class merely needs to invoke htmlGen.Start(), htmlGen.Col(*value*), and htmlGen.End() at the appropriate part of your code**
  - **There is no need to instantiate htmlGen, since all the methods are class level methods!**

Computer Exercise: Emitting HTML from Java

1. Compile htmlGen.java

2. Modify InvRept.java as follows:

   * Comment out all lines containing System.out.println...
     (suggestion: use the "//" style of comment)

   * After the instantiation of InventoryItem, add code to do the following:

     * Invoke the putStart() method from htmlGen
     * Invoke the putCol() method from htmlGen, passing
       the part number, two spaces, and the description
       as a single argument (concatenate the pieces)
     * Invoke the putEnd() method from htmlGen

3. Compile InvRept.java, and test - you should see a number of lines of HTML displayed

4. Re-run InvRept and redirect the output to a file, ***e.g.***:

   java InvRept > Page1.html

5. When the file has been created, point your browser at it.

**Exercise stretch**: Add a couple of extra putCol invocations, just passing literals, such as:

```
htmlGen.putCol("PART00318   Juniper berries");
htmlGen.putCol("PART00422   Wonderful tales");
```

then compile and test the result.

# Section Preview

- The Anatomy Lesson
  - The Anatomy of a Java application
  - The Anatomy of a compilation unit
  - The Anatomy of a class
  - Accessibility
  - The Anatomy of a field
  - The Anatomy of initializers
  - The Anatomy of a constructor
  - The Anatomy of a method
  - Accessing Class Fields and Instance Fields
  - Accessing Class Methods and Instance Methods
  - Summary and recommendations
  - Rework Case Study (Machine Exercise)

# The Anatomy of a Java Application

❒ **Every Java application consists of files stored in JAR files / packages / directories, containing classes which define the fields and methods that make up the class**

❒ **Every Java application begins with the invocation of the main() method in the top application class - this method drives the whole application, mostly through invoking other methods**

- ♦ **When a method is invoked, the class loader locates and loads the class into memory, if it is not already present**

  - ✗ If there are arguments passed, these are set up in heap storage

  - ✗ If the class has any static variables, these are initialized only when the class is loaded

    - ➢ Static variables are variables used for / by all objects in the class, such as a counter of how many objects are currently instaniated from the class, or a grand total, or some shared piece of information

  - ✗ The requested method is then run; methods can be static (run for the class as a whole) or not (run against a specific object / instance)

# The Anatomy of a Java Application, 2

- **At any point, if an object is created (using new) memory management allocates storage for the object from heap storage and runs the appropriate constructor**

  - ✗ The constructor gathers the values for the fields for the new object from an external data source, one of

    - ➢ Application command line arguments
    - ➢ Application command line system properties
    - ➢ HFS file(s)
    - ➢ Classic MVS data sets
    - ➢ Relational database
    - ➢ User interaction (terminal prompt / response)
    - ➢ Data server

  - ✗ For persistent objects, some part of the application is responsible for saving any changes to the underlying data on some external data store

    - ➢ Java **garbage collection** routines will automatically free the allocated storage used by the object once it can determine the object is no longer being referenced

- **Any time an exception occurs, any available application-supplied exception handling routines will be invoked**

# The Anatomy of a Compilation Unit

❒ **A compilation unit is a text file containing your Java code**

- ♦ **One or more class definitions (other possibilities discussed later)**

❒ **If there are two class definitions, they are nested or not:**

**Non-nested**

```
One class {
.
.
.
}
```

```
Two class {
.
.
.
}
```

**Nested**

```
One class {
.
.
.
    Two class {
    .
    .
    .
    }
}
```

- ♦ **Within a nested class, there may be a nested nested class, so to speak**
- ♦ **Within a compilation unit there may be more than two classes, some nested some not (but no overlapping class definitions)**
- ♦ **Only one non-nested class in a compilation unit may be specified as** public
- ♦ **Non-nested classes are also refered to as "top level classes" and "top level members" of a compilation unit**
  - ✗ We are only concerned with non-nested classes at the moment

# The Anatomy of a Class

❒ **Ignoring nested classes, a class definition has the following structure:**

```
package declaration
import declaration(s)

class header {

    field                    class
    field                        fields
    ...

    field                    instance
    field                        fields
    ...

    class initializers
    instance initializers

    constructors

    class methods
    instance methods

}
```

❒ **Note that every piece is optional**

- ♦ **You can have a class with no fields - methods could issue messages, accept parameters, and do other work (*e.g.*: htmlGen)**
- ♦ **You can have a class with no methods - fields might be data referenced from other classes and methods**
- ♦ **You can have an empty class - but why?**

❒ **We're done with package and import declarations for now, next we'll look at each of these other pieces in the following pages ...**

# Accessibility

❒ **A crucial question when working with Java is: when can code in a method or constructor reference fields, constructors, and methods from other classes?**

- ♦ **One wants to make needed classes available, while protecting critical data from being inadvertently changed (which could be done by either a method or a constructor)**

❒ **Generally, one sets classes to be public or to have default access**

- ♦ **Default access classes may be referenced only by classes in the same directory or in directories included in the CLASSPATH environment variable**

- ♦ **Public access classes may be referenced by the same classes as for default access (same directory and CLASSPATH directories) plus classes in the same package, and by any class that imports the package the class is in**

  ✗ So declaring a class to be **public** builds a wider audience

❒ **Note that individual constructors, fields, and methods may have their own, more restrictive accessabilty set independent of the class**

# Class Header

❒ **The next level of full syntax for declaring a class is this:**

**[public] [final] [strictfp] class** *class_name* **{ ... }**

**Where**

- **public - the constructors, fields, and methods in the class may be referenced as just discussed; if** public **is omitted, the class has default access, as described on the previous page**

- **final - this class may not have any subclasses defined on it**

- **strictfp - conversions of float and decimal values follow precise rules (default is to allow implementor some flexibility, at the [small] risk of inconsistencies across platforms)**

  ✗ We do not discuss **final** and **strictfp** any more in this course, except for the occasional passing reference, since they are of minimal use in general

# The Anatomy of a Field

❒ **The syntax for declaring and defining a field in a compilation unit is:**

**[[public] [protected] [private]]**

**[static] [final] [transient] [volatile]**

*data_type field_name* **[=** *expression***];**

**Where**

- **public - field is accessible from everywhere the containing class is accessible**

- **private - field is only accessible from the class where the field is defined ("internal use only"); it is not inherited by subclasses**

- **protected - field may be accessed only from constructors and methods in classes in the same package**

  ✗ The above attributes are called access modifiers because they allow / restrict access to a field; only zero or one of these may be specified

- **default access - if none of** public**,** protected**, nor** private **are specified, this field is available to all code in its own class, classes in the same directory or in CLASSPATH directories or classes that import the package this class is defined in**

# The Anatomy of a Field, 2

❒ **The syntax for declaring and defining a field in a compilation unit is (repeated from previous page):**

**[[public] [protected] [private]]**

**[static] [final] [transient] [volatile]**

*data_type field_name* **[=** *expression***];**

**Where, continued**

- **static - indicates this field is tied to the class, not to any particular instance (object)**
  - ✗ No matter how many instances there are of this class, there is only one instance of a static field
  - ✗ Might be useful for counting the number of instances currently extent, or for read-only text, or for a field for generating keys to a file, and so on
  - ✗ Static variables are often called class variables, and non-static fields are called instance variables
- **final - if a field is declared final, it must have a value assigned by the time any initializers have run; it may not be changed after that; both static and non-static fields may be declared final (this is like specifying a constant)**
- **transient - indicates to the system that this field does not need to be preserved for class state information (pretty obscure)**
- **volatile - used when working with threads**

# The Anatomy of a Field, 3

**Notes**

- **For *data_type* specify any primitive or reference type**

- **Class fields (static fields) need to be declared outside of any methods or instance constructors**

  - ✗ They do not need to appear before the declaration of non-static fields - static and non-static field declarations may be mixed together

- **You may optionally assign a value to a field, as shown**

- **One of the deeply felt theories of Object Oriented programming is that fields in a class or object should not be easily changed from outside the class definition**

  - ✗ Which argues for declaring most fields as **private** or **protected**

    - ➢ Then provide methods for accessing the value or changing the values

      - ➤ Such methods are called "accessors / mutators", get / set methods, getters and setters and the like

    - ➢ Recall that **private** fields may not be inherited by subclasses, so that tends to argue towards using **protected** over **private**

# The Anatomy of Initializers

- **An initializer is code that initializes values in fields or the environment for classes and methods**

- **The simplest forms of initializers are these**

  - **String CompanyName = "Gifts and Gags";  // string initializer**

  - **String[] Options = {"update", "add", "delete"};  // array initializer**

  - **for (int i = 1 ; 1<args.length; i++) {...}  // local variable initializer**

- **And an initializer can be a whole block of code**

  - **An unnamed block of code, { ... }, outside of any method or constructor is an initializer**

  - **If the code is labeled as "static", it is run when the Class Loader loads the containing class into memory**

    - ✗ Can be used to initialize a class, as opposed to a constructor, which initializes an object (instance)

  - **If the code does not have the "static" attribute, it is run each time an object is instantiated**

# The Anatomy of Initializers, 2

**Example**

```
 public  class Invoked  {

  int            objInt=1;
  static int     classInt = 1;


// class initializer
  static {
            System.out.println("Class initializer - "+
                        " run when class loaded.");
            classInt++;
            System.out.println("class Int = " +
                                        classInt);
           }


// object initializer
  {
   System.out.println("Object initializer - " +
                      " run when instance created.");
   objInt++;
  }
  .
  .
  .
```

- **Note, then, that when an instance is created, it could have a class initializer, an instance initializer, and a constructor run on its behalf**
  - **A class initializer is only run the first time the class is referenced, through accessing a class (static) variable, a class (static) method, or when the first instance of the class is created**

# The Anatomy of a Constructor

❒ **The next level of detail for a constructor header is this syntax:**

**[[public] [private] [protected]]** *class_name*(**[***parameter_list***]) {...}**

**Where**

- **public - constructor is invokable from everywhere the containing class is accessible**

- **private - constructor is available only from the body of the top level class that encloses the constructor; note that if all consructors for a class are private, the class can not be instantiated**

- **protected - constructor is accessible only from methods in classes in the same package**

  ✗ The above attributes are called access modifiers because they allow / restrict access to a constructor; only zero or one of these may be specified

- **default access - if none of** public**,** protected**, nor** private **are specified, this constructor is invokable from all classes in the same directory or in CLASSPATH directories or classes that import the package this class is defined in**

- **Note that a constructor is invoked when an instance of the class is created - this is what it means for a constructor to be accessible or invokable**

# The Anatomy of a Constructor, 2

- *parameter_list* **is a series of pairs:** *data_type parameter_name* **each pair separated by commas**

❒ **Recall that you may have multiple constructors for a single class, differentiated by the parameters they take**

- **When you instantiate a class, the correct constructor is inovked depending on what arguments are passed**

  ✗ Reminder: this is called **overloading**

❒ **If you do not code at least one constructor for a class, Java inserts an implicit call to the default no-argument constructor**

- **And this will always be invoked when an object is instantiated from this class**

# The Anatomy of a Method

❒ **The next level of detail for a method header is this syntax:**

**[[public] [private] [protected]]  [static]  [final]  [strictfp]**

*result_type  method_name*(**[***argument_list***]**) **{...}**

**Where**

- **public - method is invokable from everywhere the containing class is accessible**

- **private - method is available only from the body of the top level class that encloses the method**

- **protected - method is accessible only from methods in classes in the same package**

  ✗ The above attributes are called access modifiers because they allow / restrict access to a method; only zero or one of these may be specified

- **default access - if none of** public**,** protected**, nor** private **are specified, this method is invokable from all classes in the same directory or in CLASSPATH directories or classes that import the package this class is defined in**

# The Anatomy of a Method, 2

❒ **The next level of detail for a method header is this syntax (repeated from previous page):**

**[[public] [private] [protected]]  [static]  [final]  [strictfp]**

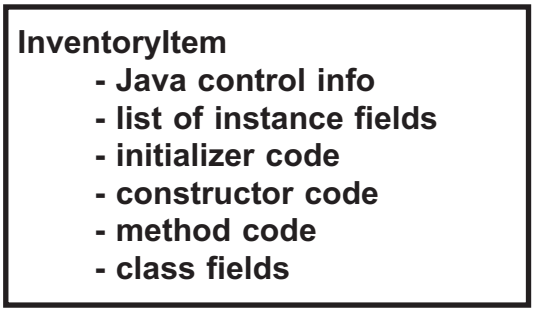
*result_type  method_name*(**[***argument_list***]**) **{...}**

**Where, continued**

- **static - indicates this method is tied to the class, not to any particular instance (object)**
  - ✗ Static methods are often called class methods, and non-static methods are called instance methods
- **final - this method may not have any subclasses override it or hide it**
- **strictfp - conversions of float and decimal values follow precise rules (default is to allow implementor some flexibility, at the [small] risk of inconsistencies across platforms)**
  - ✗ We do not discuss **final** and **strictfp** any more in this course, except for the occasional passing reference, since they are of minimal use in general

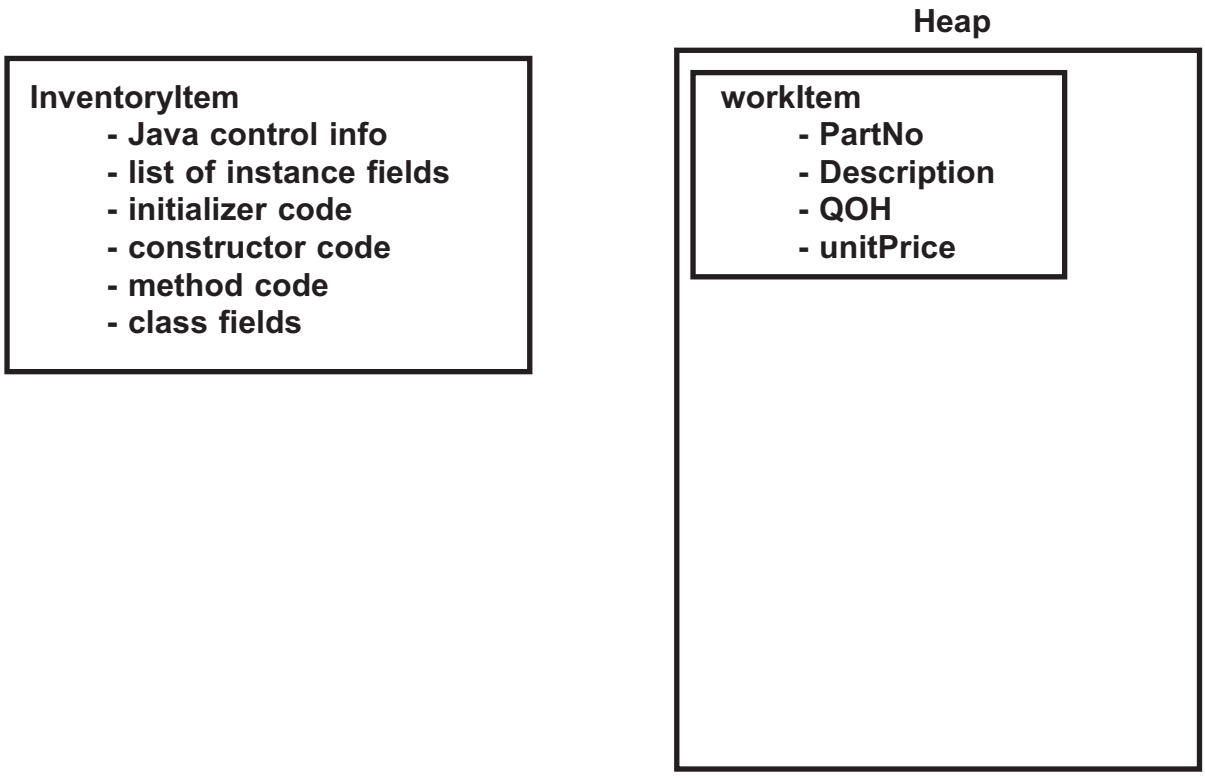
# Class Fields and Instance Fields

- **In a class definition we see there can be fields that refer to the class as a whole - these are declared as** static **and are called class fields**
  - **Fields declared in a class that are not static are instance fields**
  - **A class field might contain, for example, the next item number to assign to a new inventory item; or the total value of items in inventory**
  - **When a class file is loaded, the class loader program builds an area to hold information about the class and runs any class initializers**



# Class Fields and Instance Fields, 2

- As the application runs, when an instance of a class is created, memory is allocated from heap storage to hold the instance fields values
  - Each time an instance is created, instance initializers (if any) are run
  - And a constructor is run (remember, if no constructor is supplied or if no supplied constructor has a signature that matches the arguments passed with new, the default constructor is run)



# Accessing Class Fields and Instance Fields

❒ **In your application, if a class is accessible you access class (static) fields using the syntax**

*class_name.field_name*

- **Or possibly**

  *package_name.class_name.field_name*

- **Of course, if the field is private or protected you will have to invoke a method to get / set the value of the field**

❒ **If a class is accessible you access instance fields using the syntax**

*object_ref_name.field_name*

- **Or possibly**

  *package_name.obj_ref_name.field_name*

- **Of course, if the field is private or protected you will have to invoke a method to get / set the value of the field**

- **Also, instance fields may not be accessed from class initializers or class methods in the same class**

  ✗ While class fields may be accessed from instance initializers, constructors, and instance methods

# Accessing Class Methods and Instance Methods

❒ **In your application, if a class is accessible you <u>invoke class (static) methods</u> using the syntax**

*class_name*.*method_name*(*arg(s)*)

- **Or possibly**

  *package_name*.*class_name*.*method_name(arg(s)*)

- **Of course, if the method is private or protected you may only invoke it under the appropriate restrictions (invoke from top level class that encloses the method, or from same package)**

❒ **If a class is accessible you <u>invoke instance methods</u> using the syntax**

*object_ref_name*.*method_name*(*arg(s)*)

- **Or possibly**

  *package_name*.*obj_ref_name*.*method_name(arg(s)*)

- **Of course, if the method is private or protected you may only invoke it under the appropriate restrictions (invoke from top level class that encloses the method, or from same package)**

- **Also, instance methods may not be accessed from class initializers or class methods in the same class**

  ✗ While class methods may be accessed from instance initializers, constructors, and instance methods

# Accessing Class Methods and Instance Methods, 2

❒ **For example, note that a class method may not invoke an instance method in the same class as the static method**

- ♦ **Put another way, a static method may not invoke a non-static method in the same class**
  - ✗ So either make the non-static method "static", or create an instance of the containing class, for example:

```
class Invoker {
  .
  .
  .
 public static void main(String[] args) {
  .
  .
  .
  Invoker Copper = new Invoker();
  Copper.subber();

 }   // end of main method

void subber() {   // non-main, non-static method
.
.
.   }    // end of subber method

}  // end of class
```

- ♦ **In the above example, code in main() cannot access the method subber() directly, but it can invoke subber() as it refers to an instance, say, Copper**
  - ✗ Since subber() is an instance method (not static)

# Summary and Recommendations

- **Generally declare classes as** public **- provides wide availability**

- **Declare class fields as** public **or with default access - allows access to the fields by any code that can access the class**

- **Declare instance fields as** protected **- this allows inheritance but only allows access by way of a method**

- **Assign initial values to variables in the declaration, if possible - generally thought to be good programming practice**

- **Use a block initializer when you have many constructors that have common code**

  - **Put the common code in the initializer and each constructor should just have the code needed for its special case**

- **The access modifier for methods depends on where the containing class will reside (if it needs packages, if it should be accessible from multiple applications, and so on)**

Computer Exercise: Rework our Case Study

For this short lab, we'll tighten up our various declares in InventoryItem and add some more methods:

1. Change InventoryItem.java as follows:

   Declare the class as **public**
   Declare the four fields as **protected**
   Declare the constructor and four methods to be **public**
   In addition to the four "get" methods, create four "set" methods:

   ```
   public void setPno(String inPno)
   public void setDesc(String inDesc)
   public void setQOH(int inQOH)
   public void setUnPr(float inUnPr)
   ```

   * where each method simply has an assignment of the incoming argument to the appropriate instance field.

   * remember this is in your inventory subdirectory

2. Compile InventoryItem.java

3. Compile InvRept.java (remember, this is in your base directory); you might have to run your setJavaPackagePath script

4. Test the resulting InvRept class.

This page intentionally left almost blank.

# Section Preview

- **The Power of Subclasses**
  - **Defining a subclass**
  - **Field Hiding**
  - **Method Hiding and  Overriding**
  - **The Power of subclasses**
  - **Scope**
  - **Shadowing**
  - **Obscuring**
  - **this**
  - **super**
  - **Defining a subclass (Machine Exercise)**

# Defining a Subclass

- **Eariler (pp. 67-81) we defined our first classes for our case study: InventoryItem and InvRept**
  - **And we also mentioned the concept of inheritance (pp. 43-47) in general and theorized about defining a subclass named Foodstuffs (p. 46) which added an expiration date to the InventoryItem class**
- **In this section, we will specify how to define a subclass of our InventoryItem class, and how to work with subclasses in general**
- **First, a review of the status of InventoryItem:**
  - **Located in the inventory directory, and part of the package "inventory"**
  - **Defines four protected fields as instance variables: PartNo, Description, QOH and unitPrice**
  - **Defines a single constructor that requires all four fields**
  - **Defines eight methods, a** get **and a** set **for each of the four fields**

# Defining a Subclass, 2

❒ **To create a class, Foodstuffs, that is a subclass of InventoryItem, we declare the class with the "extends" option:**

**public class** *class_name* **extends** *parent_class name* **{ ... }**

- **Specifically, in our case, we would code:**

```
public class Foodstuffs extends InventoryItem {
```

❒ **With just this much code, we now have class Foodstuffs that has, by default, all the fields and methods of InventoryItem available**

- **But not its initializers or constructors - these are not inherited**

  ✗ Actually, if you do not specify a constructor in the child class, it will get the default no-argument constructor and not compile! - unless the parent class has some no-argument constructor

  ✗ Each constructor in the child class must extend some constructor in the parent class (that is, each child constructor's signature must match the signature of some parent class's constructor, with the possible addition of fields found only in the child class)

  ✗ When an object is instantiated, the parent's relevant initializers (if any) are run, then the parent's appropriate constructor is run (which might trigger the parent's parent's initializer(s) and constructor, and so on), then any relevant child intializers then the relevant child constructor

❒ **Of course, we need to add something to make Foodstuffs different from its parent (otherwise, why bother?) ...**

# Defining a Subclass, 3

❒ **As we mentioned before, the significant difference is that food items typically have an expiration date that is important, so we can now code, in the file Foodstuffs.java:**

```
package inventory;
public class Foodstuffs extends InventoryItem {

    private String expDate;
```

♦ **Note that ultimately we probably want to work with a real "date" kind of item, but this will take some getting to**

✗ For now, we just work with date as a String, in order to focus on the topic at hand

# Defining a Subclass, 4

❒ **Next we'll want to add a constructor**

- **As mentioned, we must build on some constructor already in InventoryItem; for our constructor header we might code:**

```
public Foodstuffs(String pno, String desc,
   int qty, float unprice, String exdate) {
```

  ✗ Which has the same signature as InventoryItem, except we have added the new field "exdate" for the instantiater to pass an expiration date

- **Clearly we want the first four fields to be treated just like in the parent class, the superclass, so we can use the** super **notation:**

```
public Foodstuffs(String pno, String desc,
   int qty, float unprice, String exdate)
{
   super (pno, desc, qty, unprice);
   expDate = exdate;
}
```

- **... and we have our constructor!**

❒ **Note: if a constructor in a subclass does not explicitly invoke a superclass constructor, Java will automatically insert a call to the no-argument constructor; if the parent class does not have a no-argument constructor, you will get a compile error**

- **Furthermore, if you invoke a superclass instructor, the** super **statement must be first in the constructor**

# Defining a Subclass, 5

❒ Again, because of inheritance, we can use the existing methods of our parent class, and simply supply a get / set pair for our new field, and our entire Foodstuffs.java file looks like this:

```
   package inventory;
// Declaration of class Foodstuffs
  public class Foodstuffs extends InventoryItem {

     protected String expDate;

    // Definition of constructor

        public Foodstuffs(String pno, String desc,
                int qty, float unprice, String exdate)
        {
           super (pno, desc, qty, unprice);
           expDate = exdate;
        }


    // Definition of method getExpDate
        public String getExpDate()
        {
           return expDate ;
        }


    // Definition of method setExpDate
        public void setExpDate(String inDate)
        {
           expDate = inDate;
        }

 }   // end of class definition Foodstuffs
```

# Defining a Subclass, 6

❒ **Now any method may reference Foodstuffs instances using these constructs:**

♦ **Create a new instance**

```
Foodstuffs workFood = new
   Foodstuffs("PART00410", "Fogwish", 9, 32.50f,
         "20071010");
```

✗ Or other forms of available constructors

♦ **Access any of the fields using available methods:**

```
System.out.println("Part number is "
    + workFood.getPno());

System.out.println("Description = "
    + workFood.getDesc());

System.out.println("Quantity on Hand = "
    + workFood.getQOH());

System.out.println("Unit price = "
    + workFood.getUnPr());

System.out.println("Expiration date is "
    + workFood.getExpDate());
```

✗ Similarly for the "set" methods, of course

# Defining a Subclass, 7

❒ **Some rules / terms relating to subclassing:**

- ♦ **A class may only extend one other class (single inheritance)**
- ♦ **Every class has only one direct parent (direct superclass), except the class called Object, which is the starting point for all classes and has no direct parent**
- ♦ **A class may have any number of children (child classes, subclasses)**
- ♦ **By default, a child class inherits all the fields and methods of its direct parent (both class and instance fields and methods)**
  - ✗ It may then override any method by redefining it (details soon)
- ♦ **However, fields and methods with** private **or default access are not inherited if the subclass is in a different package than its direct parent class**
  - ✗ Methods declared as **private** are not inherited even if the parent and child classes are in the same package
- ♦ **Classes declared as** final **may not be extended**

# Field Hiding

❒ **In a subclass, declarations of fields that have the same name in both the parent and child classes suppresses inheritance and causes something called field hiding**

❒ **Suppose, for example, there are two fields that are defined in both InventoryItem and Foodstuffs, maybe:**

```
public static float MaxPrice;   (<-- class)
public String marker;           (<-- instance)
```

♦ **So, if a field name is declared in both the parent and the child ...**

✗ In code in the child class (initializers, constructors, or methods), a simple reference to the field is assumed to be a reference to the child's version (instance field or class field)

➢ In Foodstuffs, **MaxPrice** refers to the value associated with the Foodstuffs class, and **marker** refers to the value in the current instance of Foodstuffs

✗ In the child, a reference to **super.***field_name* is a reference to the parent's version of the variable (instance field or class field)

➢ In Foodstuffs, **super.MaxPrice** refers to the value associated with the InventoryItem class, and **super.marker** refers to the value in the current instance of InventoryItem

# Field Hiding, 2

✗ The parent is unable to reference a child instance field, but can reference a child's class field by *child_class_name.field_name*

➢ In InventoryItem, a reference to **Foodstuffs.MaxPrice** is a reference to the value in Foodstuffs

✗ In an outside class, a reference to child_object_name.field_name, is the child's version of an instance variable

➢ In InvRept, a reference to **workFood.marker** references the value of marker in workFood

✗ In an outside class, to reference the parent's version of a class field, you can **cast the object to the class of the parent(!),** for example:

```
workFood.MaxPrice
```

➢ references the Foodstuffs's variable, while

```
((InventoryItem)workFood).MaxPrice
```

➢ references the InventoryItem's variable

✗ You may also access the parent's version of static variables directly

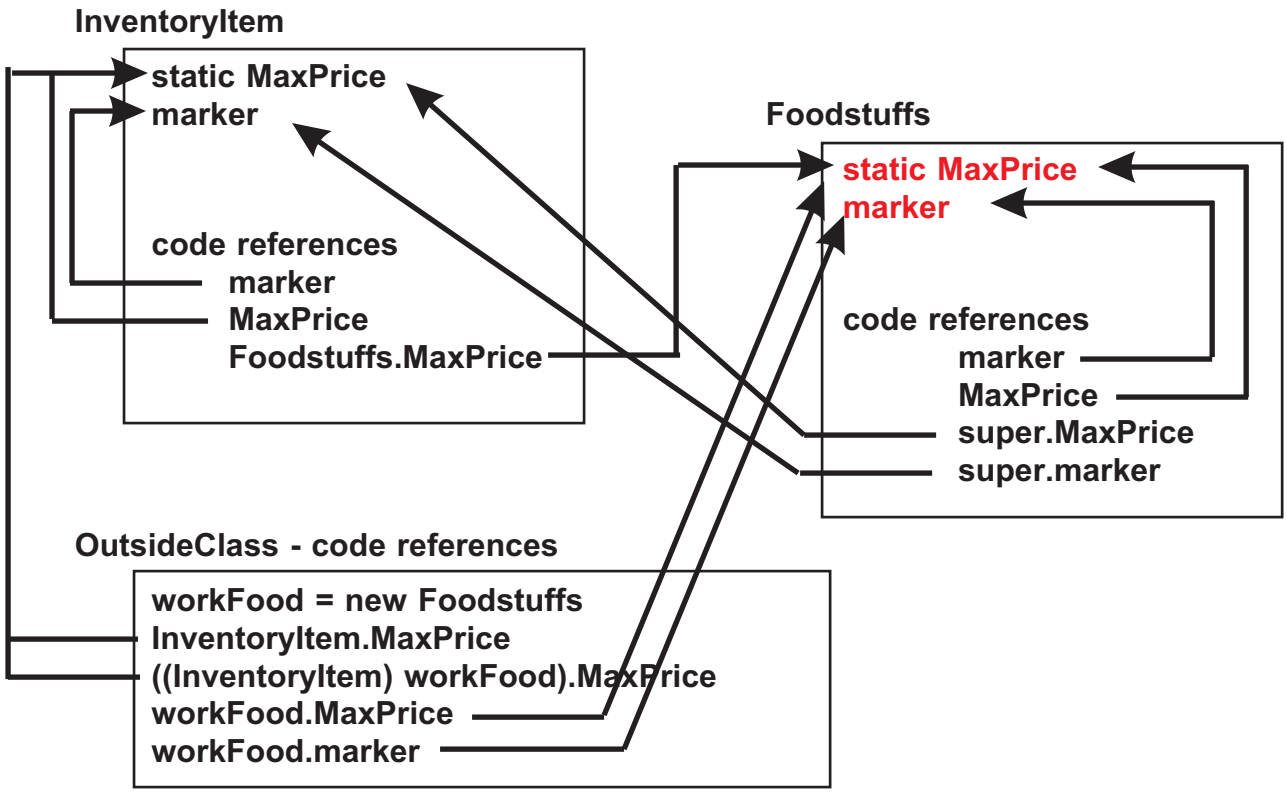
➢ For example, in InvRept, **InventoryItem.MaxPrice** accesses the InventoryItem class variable as opposed to **Foodstuffs.MaxPrice**

# Field Hiding, 3

- A rather unexpected side effect here is that while the child does not inherit the variable marker (because of hiding), the child implements it: instances of Foodstuffs have two versions of the variable that are available with them

- Note that a field will hide any field of the same name in its parent class, even if the fields are of different types

- Generally, field hiding is discouraged
  - It can lead to unexpected results
  - It can make it harder to code data references



# Method Hiding and Overriding

❒ **Methods in subclasses work a little different than fields in subclasses**

- ♦ **A static method (class method) with the same name as a static method in the parent class hides the method in the parent**
  - ✗ Both methods are available to the subclass, with appropriate prefixing
- ♦ **An instance method with the same name as an instance method in the parent class overrides the method**
  - ✗ In the subclass, the parent instance method is not available
- ♦ **There are some subtleties here ...**
  - ✗ You can't override class methods, only instance methods
  - ✗ A field hides any field of the same name, regardless of data type, but a method hides or overrides only methods with the same signature
  - ✗ The details ...

# Method Hiding and Overriding, 2

❒ **Again, an example; suppose InventoryItem has these methods:**

```
public static classShow() {
   System.out.println("Class from Inventory"); }

public   dataShow() {
   System.out.println("Instance from Inventory"); }
```

♦ **And Foodstuffs has these methods:**

```
public static classShow() {
   System.out.println("Class from Food"); }

public   dataShow() {
   System.out.println("Instance from Food"); }
```

♦ **Finally, assume workItem is an InventoryItem object and workFood is a Foodstuffs object; then ...**

# Method Hiding and Overriding, 3

❒ **Invoking these methods produces these behaviors:**

- ♦ **Within either class, invoking one of these methods gets that class's version**
- ♦ **From an outside class, you see these behaviors:**
  - ✗ **InventoryItem.classShow(); displays** Class from Inventory
  - ✗ **Foodstuffs.classShow(); displays** Class from Food
  - ✗ **workItem.dataShow(); displays** Instance from Inventory
  - ✗ **workFood.dataShow(); displays** Instance from Food
  - ✗ **workFood.classShow(); displays** Class from Food
  - ✗ **workItem.classShow(); displays** Class from Inventory
  - ✗ **InventoryItem.dataShow()** and **Foodstuffs.dataShow()** are not allowed (cannot invoke instance method from class reference, or in the language of the compiler, "non-static method dataShow() cannot be referenced from a static context")

❒ **Pretty vanilla, and what you'd expect**

# Method Hiding and Overriding, 4

❒ **Now we take a bit of a leap with this code:**

```
InventoryItem workWhat = workFood;
```

- ♦ **Notice the absence of "new" here**

- ♦ **This says "workWhat" is a class variable that can hold a reference to objects in InventoryItem or any subclass of InventoryItem**

  - ✗ And for right now, "workWhat" should contain the same reference as "workFood"

**Behaviors**

- ✗ **workWhat.dataShow();** returns **Instance from Food**

- ✗ **workWhat.classShow();** returns **Class from Inventory**

  - ➢ In this sense, the overridden instance method dataShow() in InventoryItem is not accessible, but the hidden class method classShow() in InventoryItem may be revealed and invoked

  - ➤ This works this way because "workWhat" was declared as type "InventoryItem"

# Method Hiding and Overriding, 5

- **Recall that a field hides any other field with the same name, regardless of type**
  - **It's a little different with hiding of methods: a method only overrides or hides a method of the same type (instance or class) with the same signature**

- **Finally, hidden methods can be overloaded**
  - **For example, suppose there is a method, dataShow(float) declared in InventoryItem - it has the same name as the method in the child, but a different signature; we see:**
    - ✗ **workItem.dataShow(12.25f); displays** Instance from Inventory
    - ✗ **workFood.dataShow(12.25f); displays** Instance from Inventory
    - ✗ **workWhat.dataShow(12.25f); displays** Instance from Inventory
  - **That is, even though dataShow() in InventoryItem is overridden by dataShow() in Foodstuffs, dataShow(float) has a signature only found in InventoryItem, so it is not overridden**
  - **Foodstuffs can contain overloaded methods also**

# The Power of Subclasses

- **Now we're in a position, finally, to get an idea of what all the shouting's about, regarding inheritance and code reuse ...**

- **Suppose I want to add a new instance method: calcValue**

  - **This routine takes, for the current instance, the quantity on hand (QOH) times the unit price (unitPrice), which tells us the total value of that item**

  - **The code is straightforward, something like this:**

```
public float calcValue() {
   return (QOH * unitPrice);
}
```

  - **Then if I put this method in InventoryItem, in our InvRept code (or in any class that accesses InventoryItem) we can code:**

```
System.out.println("Value is " +
     workItem.calcValue());
```

  - **But the neat thing is that Foodstuffs automatically inherits this method!**

    - ✗ So invocations of workFood.calcValue() also work on Foodstuffs objects

# Scope

- Every name (field, constructor parameter, method parameter, local variable, and so on) has a scope from the point where it is declared
  - A range of statements where the name is correctly recognized and can be accessed by its simple name
  - Of the kinds of names that have scope, we have so far examined fields, constructor parameters, method parameters, and local variables
  - To set the stage for the next discussion we demonstrate examples of these kinds of things and their scope ...

# Scope - Fields

```
// Declaration of class InventoryItem
  public class InventoryItem {

     String PartNo;
     String Description;

     // Definition of constructor methods

        public InventoryItem()
         {
            PartNo = "Part0000";
            Description = " ";
         }

        public InventoryItem(String pno, String desc)
         {
            PartNo = pno;
            Description = desc;
         }

    // Definition of method getPno
       public String getPno()
         {
           return PartNo ;
         }

    // Definition of method setDesc
        public void setDesc(String inDesc)
        {
           Description = inDesc;
        }
 }   // end of class definition InventoryItem
```

- The scope of instance and class fields is the whole program they are declared in (PartNo and Description in our example)

# Scope - Constructor Parameters

```
// Declaration of class InventoryItem
  public class InventoryItem {

    String PartNo;
    String Description;

    // Definition of constructor methods

       public InventoryItem()
        {
           PartNo = "Part0000";
           Description = " ";
        }

       public InventoryItem(String pno, String desc)
        {
           PartNo = pno;
           Description = desc;
        }

   // Definition of method getPno
      public String getPno()
       {
          return PartNo ;
       }

   // Definition of method setDesc
       public void setDesc(String inDesc)
       {
          Description = inDesc;
       }
 }   // end of class definition InventoryItem
```

- The scope of constructor parameters (pno and desc in our example) is the whole constructor

# Scope - Method Parameters

```
// Declaration of class InventoryItem
  public class InventoryItem {

    String PartNo;
    String Description;

    // Definition of constructor methods

      public InventoryItem()
       {
         PartNo = "Part0000";
         Description = " ";
       }

      public InventoryItem(String pno, String desc)
       {
         PartNo = pno;
         Description = desc;
       }

   // Definition of method getPno
     public String getPno()
      {
        return PartNo ;
      }

   // Definition of method setDesc
      public void setDesc(String inDesc)
      {
        Description = inDesc;
      }
 }  // end of class definition InventoryItem
```

- The scope of method parameters (inDesc in our example) is the whole method

# Scope - Local Variables

❒ **Local variables are those defined inside a method or code block or the initialization part of a** for **or** enhanced for**:**

```
  public static void main(String[] args) {

     int[] arr1    = {1, 2, 3};
     int[] arr2    = {4, 5, 6};
.
.
.
           {int stepper = 0;
            String transCodes = "acdq";
             .
             .
             .
           }
```

♦ **Scope of local variables (arr1, arr2, stepper, transCodes above) is the whole block containing the declaration**

```
  for (int cntr=0; cntr < top5.length; cntr++)
        {
         System.out.println("cntr = " + cntr +
            ", top5[" + cntr + "] = " + top5[cntr]);
        }
```

♦ **For** for **and** enhanced for**, scope of local variables (cntr above) begins at the declare and includes the other parts of the initialization / condition / and update parts of the statement plus the statement(s) in the enclosed block**

# Shadowing

- **Sometimes a name is declared inside the scope of an already declared name**
  - **Usually the Java compiler considers this an error**
  - **But other times, the declaration is allowed because the scope of the new name is limited and clear - this is called shadowing:**
    - ✗ The newly declared item is in effect in its limited scope, and when that scope is exited, the shadowed item is restored to be the item associated with that name

# Shadowing, 2

## Example

- If we have a class field declared, then declare a field with same name in a method in that class, both variables can work their magic:

```
public class DisplayList {

    static int cntr = 7;

  public static void main(String[] args) {

     String[] top3 = {"Hope", "Humor", "Health"};

     System.out.println("cntr = " + cntr);

     System.out.println("Top 3 choices are ...");
     for (int cntr=0; cntr < top3.length; cntr++)
        {
         System.out.println("cntr = " + cntr + ",
            top3[" + cntr + "] = " + top3[cntr]);
        }

     System.out.println("cntr = " + cntr);
  }
}
```

- Note the inner "cntr" is shadowing the outer "cntr"; when this code is compiled and run, the result is:

```
cntr = 7
Top 3 choices are ...
cntr = 0, top3[0] = Hope
cntr = 1, top3[1] = Humor
cntr = 2, top3[2] = Health
cntr = 7
```

# Obscuring

- A simple name could occur in a context where it is not clear if the name represents the name of a variable, a type, or a package
  - That is, the simple name occurs as two or more of these and which to use it as is ambiguous
  - Java rules say that in such cases the compiler will decide which possible meaning to apply in this order: variable name, type name and package name

  For example

  - In our previous DisplayList class, we could add this field declaration:

```
static String DisplayList = "Data string";
```

  - Then in our main() method code:

```
System.out.println("DisplayList = " +
                   DisplayList);
```

  - Since the name could be interpreted as either a class name or a field name, the rules choose to interpret it as a field name

- Potential problems that could arise from hiding, shadowing, and obscuring can be largely avoided with tight enforcement of a well-defined naming convention

# this

- The keyword "this" can be used as a prefix to a field in the context of an instance method, an instance initializer, or a constructor
  - It is a way to say "I mean the version for the current instance"
  - A common example is a constructor like this:

```
public InventoryItem(String PartNo,
                     String Description)
  {
     this.PartNo = PartNo;
     this.Description = Description;
  }
```

  - Notice that this allows us to use the field name for the constructor parameter name also, instead of having to create new names
  - Similarly used in methods

# super

❒ **The keyword "super" can be used as a prefix to indicate a reference to the immediate superclass of the current class**

- **We saw an example earlier, on page 161**

  ✗ In general, for methods and constructors this is a good way to take advantage of reusing the code in the superclass

  ✗ For initializers and constructors, the syntax is:
  super (*parameter_list*);
  which says invoke the superclass constructor or initializer whose signature matches *parameter_list*

- **Sometimes you can use super.*field_name* to access a hidden field, depending on how its hidden**

- **You can use super.*method_name*(*args*) to access an overridden method**

- **Initializers for both class and instance variables can find this useful**

- **Of course, "super" only applies in the context of a subclass (child class)**

- **If a method has only default access, then "super" only finds the method if the parent class and child class are in the same package**

## Computer Exercise: Defining a Subclass

In your J510 directory is a file named Foodstuffs.java. This file contains the defintion of the Foodstuffs subclass as shown on page 162. Move this file to your inventory subdirectory and compile it.

Still in your inventory subdirectory, modify InventoryItem.java to have a new instance method, calcValue, as described on page 173. Recompile InventoryItem.java.

Returning to your J510 directory, in your InvRept.java file, add some code to:

* invoke htmlGen.putCol passing this string:
  Value for current item = testItem.calcValue()

* create a Foodstuffs instance, testFood, using, say:
  new Foodstuffs("PART00410", "Fogwish", 9, 32.50f, "20071010");

* invoke htmlGen.putCol passing the string
  testFood.getPno() + "  " + testFood.getDesc()

* invoke htmlGen.putCol passing this string:
  Value for current Food = testFood.caclValue()

Compile and run InvRept.java; when it's running successfully, re-run it with the output being routed to Page1.html as before; point your browser at this page.

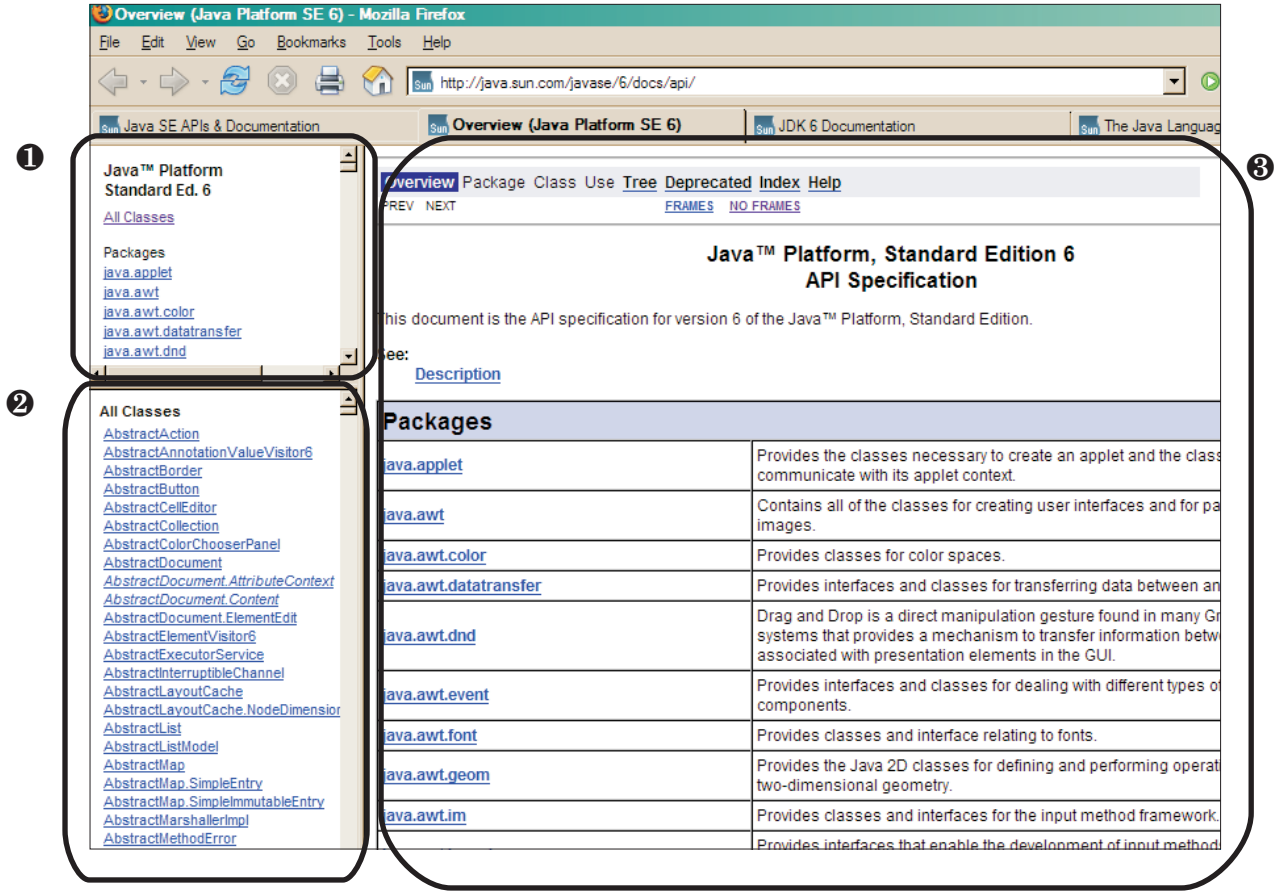
# Section Preview

- Wrapper Classes
  - Java Docs - the API Specification
  - Wrapper Classes
  - Abstract Classes and Methods
  - The Number class
  - The Integer wrapper class
  - Boxing and Unboxing
  - Wrapper Classes (Machine Exercise)

# Java Docs - The API specification

❒ **In an earlier section we examined two of the major sources for official information about Java - here we examine a third:**

♦ **The link http://java.sun.com/javase/6/docs/api/ gets you to this page:**




Let's Do It!

Computer Experiment: Get to this page in your browser, take a little time looking around (postpone if not possible)

# Java Docs - The API specification, 2

❒ **This is a very information rich window, with three inter-related panels:**

- **1 - The packages panel: What package(s) to look at; the default is all packages that come with java; leave as is or select one specific package**
- **2 - The classes panel: Of the selected package(s) a list of all the classes available in that package / those packages**
- **3 - The details panel: Once a class is selected this panel contains the details for that class, with many parts:**
  - ✗ Synopsis of the class (name, package, parent class, interfaces implemented, brief discussion)
  - ✗ Field summary - list of each field in the class and its characteristics
  - ✗ Constructor summary - syntax and one line description for each constructor available
  - ✗ Method summary - syntax and one line description for each method available; list of inherited methods
  - ✗ Field detail - deeper discussion of each field in the class
  - ✗ Constructor detail - deeper discussion of each constructor
  - ✗ Method detail - deeper discussion of each method

# Java Docs - The API specification, 3

- Knowing how to read the API specification is the key to being self-reliant in Java, so we will spend some quality time rummaging around this location
  - Examining a couple of classes in great detail, with lots of examples
- In particular, we examine the java.lang package, and in there we look at the classes: Integer and Number
  - With these under our belt, we should be able to pick up just about any class write up and understand how to use it

So here we go ...

[If you're following along, select java.lang in the packages panel and examine the classes from that package listed in the classes panel ]

# Wrapper Classes

❒ **The data types that Java classifies as "primitive" have only a name (location in memory) and a value**

- ♦ **There are no class facilities available to do conversion, formatting, and so on**

❒ **So Java provides a set of wrapper classes, one for each primitive type, that provide methods to work with data of that type**

| Primitive type | Wrapper class |
|---|---|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

❒ **Integer seems like as good a class to examine as any**

- ♦ **But when you get there, you find that Integer is a subclass of Number - take a look at the Integer class, if you can -**

  - ✗ So, in order to get the real story, we should examine Number first - so we go there and find ...

# The Number Class

- **The class called Number is a subclass of Object; Object is its direct parent, its superclass**

- **Number is an abstract class - this is a technical term that means at least some of the methods of the class are declared but not defined: the defining must be done in the subclasses of the abstract class**

  - **Byte, Double, Float, Integer, Long, and Short are all subclasses of Number**

- **Number is also public, and since java.lang is implicitly always imported, the Number class constructors, and methods are always available**

  - ✗ Note that the Number class has no fields, just methods: the purpose is to provide a basis for interfaces and methods

  - ✗ And Number has only the empty constructor

- **Number implements the interface called Serializable; since we haven't talked about interfaces yet, we will just move on**

  - **But first, a slight digression on abstract classes and methods ...**

# Abstract Classes and Methods

❒ **The primitive types byte, short, int, long, float, and double are all numeric types**

- ♦ **So it seems reasonable that a Numeric class should have methods for converting between these numeric types, whenever possible**

❒ **And we'll see shortly that the Number class indeed has methods such as byteValue(), doubleValue(), and so on**

- ♦ **But the Number class itself would have a tough time implementing such methods for all numeric types**

  - ✗ Instead, why not let each specific Number subclass implement these methods as appropriate for its respective number type?

❒ **So the methods in the Number class have a new attribute in their declaration:** abstract

❒ **When a method is declared as abstract, it has no code tied to it: it is expected that the subclasses that use these methods will include the code to implement them, as appropriate for each subclass**

# Abstract Classes and Methods, 2

- **The rationale behind abstract classes, in general, is that you may have a number of classes with common processing requirements but each has need for some unique methods too**
  - **Consider a simple class such as "mailing labels": turns out there are very different issues depending on country, business vs. personal, size of label, and so on**
  - **Yet there are many common parts, too, such as addressee name and street address and city**
- **So you could define a class that includes the code for the full definition of commonly shared methods and just the declarations for abstract methods, for the methods that are implemented differently**
  - **For example, a MailingLabel class would include actual code for assembling the addressee name, street address, and city into fields**
  - **Then create separate subclasses for each distinct type of mailing label you need**
    - ✗ Each extends MailingLabel and implements all the abstract classes in the style required for that subclass
  - **So you can reuse the common methods and implement the unique methods separately**

# Abstract Classes and Methods, 3

❒ The Number class allows you to declare a variable that can hold any kind of numeric value, for example:

```
Number myNum;
myNum = 1;       // byte   (integer size implied
myNum = 5212;    // short   by the size
myNum = 42777;   // int      of the value)
myNum = 2L;      // long
myNum = 3f;      // float
myNum = 4d;      // double
```

- ♦ Notice this is done without instantiation

❒ The only methods available for Number are

- ♦ byteValue() - return specified number as a byte
- ♦ doubleValue() - return specified number as a double precision floating point number
- ♦ floatValue() - return specidifed number as a single precision floating point number
- ♦ intValue() - return specified number as an integer value
- ♦ longValue() - return specified number as a long integer
- ♦ shortValue() - returns specified number as a short integer

❒ When you invoke one of these methods against a variable of type Number, Java knows the type of data currently in the variable and performs the correct conversion

# Abstract Classes and Methods, 4

❒ **Here's an example of an abstract method declaration:**

```
abstract public byte byteValue();
```

**Notice:**

- **Semi-colon at end of declare, and no braces-bound block of code for the method definition**

# Abstract Classes and Methods, 5

## General rules for abstract classes and methods

- **An abstract method must not have a definition**

- **If a class has one or more abstract methods, the class itself must be declared as abstract,** *e.g.*:

```
abstract public class MyNumber { ... }
```

- **An abstract class may have both abstract and non-abstract methods, but if all methods are abstract, you should use an interface instead of an abstract class**

  ✗ For the Numeric class, byteValue() and shortValue() are non-abstract: they are implemented as standard methods of the class

- **An abstract class might not have any abstract methods**

- **An abstract class may not (is not allowed to) be instantiated**

  ✗ However, a variable of the type of an abstract class can be correctly initialized with a reference to any [non-abstract] subclass (as shown on page 193; see also page 25)

- **The abstract methods of an abstract class must be defined by all its subclasses that are not themselves abstract**

# The Number Class, 2

❒ **The Number class has the following methods:**

- **byteValue() - return specified number as a byte**
- **doubleValue() - return specified number as a double precision floating point number**
- **floatValue() - return specified number as a single precision floating point number**
- **intValue() - return specified number as an integer value**
- **longValue() - return specified number as a long integer**
- **shortValue() - returns specified number as a short integer**

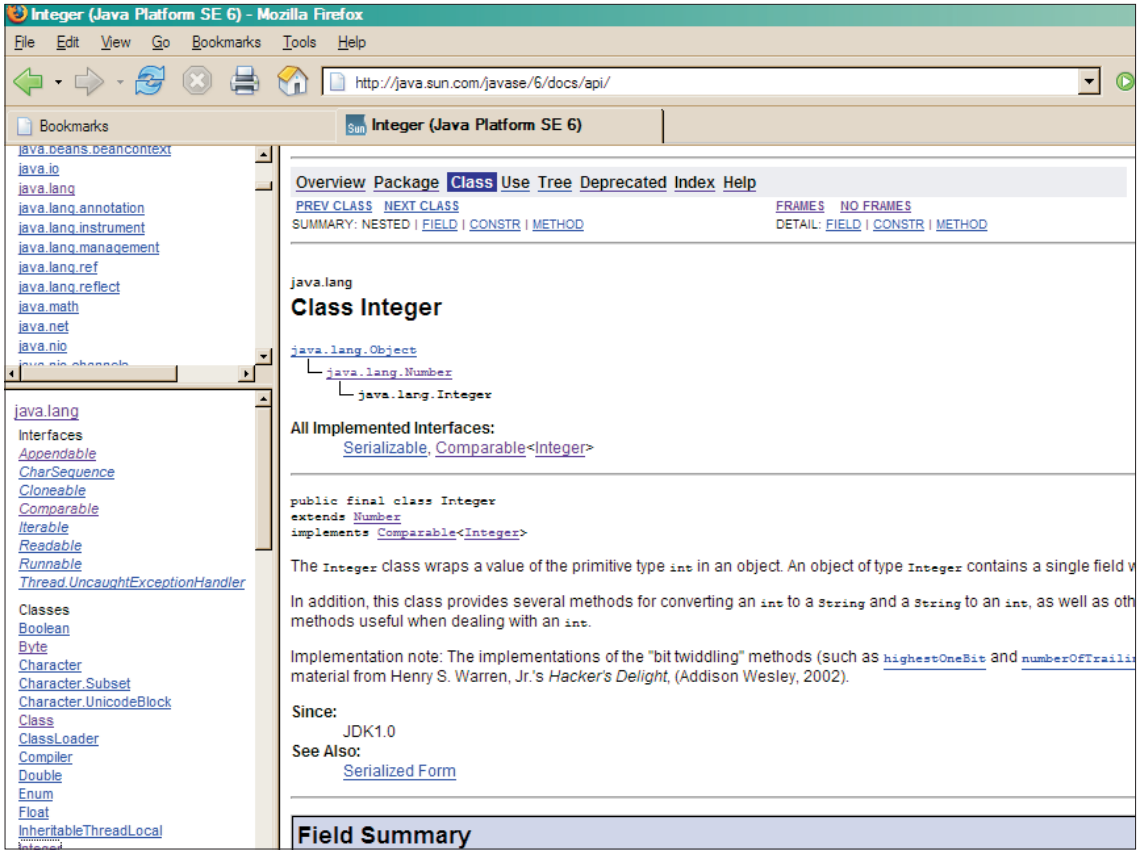
❒ **What does it mean above, "the specified number"?**

- **When the method is invoked, it will be on an object of a subclass of Number (remember, abstract classes cannot be instantiatied), in the format** *object_name.method_name()***,and the object will have a numeric value to work on**

  ✗ Maybe, for instance workNum.doubleValue()

# The Integer Wrapper Class

- ❒ **Finally (as far as this section goes) we explore the Integer wrapper class provided with Java (follow along the online docs if you can)**
  - ♦ **If you look at the documentation for the Integer class, you see this to start:**




- ❒ **First we observe that Integer is a subclass of Number - so it must implement all of Number's abstract methods (and it inherits the rest)**
- ❒ **Again, we skip over the interfaces part of the write up for now**
- ❒ **Next notice that Integer has the final attribute, which means you cannot create a subclass of Integer(!)**

# The Integer Wrapper Class, 2

❒ **There are four class level fields belonging to the Integer class**

- **MAX_VALUE, MIN_VALUE - defining the allowed range of values an integer may contain; usage example:**

```
System.out.println("Integer.MAX_VALUE = "
                   + Integer.MAX_VALUE);
System.out.println("Integer.MIN_VALUE = "
                   + Integer.MIN_VALUE);
```

  ✗ When run produces:

```
Integer.MAX_VALUE = 2147483647
Integer.MIN_VALUE = -2147483648
```

- **SIZE - returns the number of bits to hold an int value, always 32**

- **TYPE - returns the name of the primitive type the Integer class represents, always** int

# The Integer Wrapper Class, 3

❒ **There are two constructors to choose from when instantiating an Integer object:**

- ♦ **Pass an integer value (variable or literal)**

- ♦ **Pass a String value (variable or literal)**

**Examples**

```
Integer counter1 = new Integer(f3);
Integer counter2 = new Integer(3);
Integer counter3 = new Integer(s3);
Integer counter4 = new Integer("3");
```

✗ All produce objects with the related int value being 3, assuming **f3** is declared an int and has a value of 3 at the time the above statements are issued, and that **s3** is declared as String and has a value "3"

- ♦ **And, of course, an object variable can be declared of type Integer:**

```
Integer work_int;
```

✗ Always keep in mind the distinction between a class and an object (instance)

# The Integer Wrapper Class, 4

❒ **There are 34 methods defined for the Integer class**

- ♦ **Along with eight methods inherited from its ancestor class of Object, which we will not be examining here**

❒ **byteValue(), doubleValue(), floatValue(), intValue(), longValue(), shortValue() are the methods inherited from Number (the first and the last were non-abstract, the others all abstract methods)**

- ♦ **Each returns a value in the type implied, converting the specified Integer object, truncating as necessary, <u>for example</u>, given:**

```
Integer work_int1 = new Integer(37);
Integer work_int2 = new Integer("23678");
Integer work_int3 = new Integer("-1215");
```

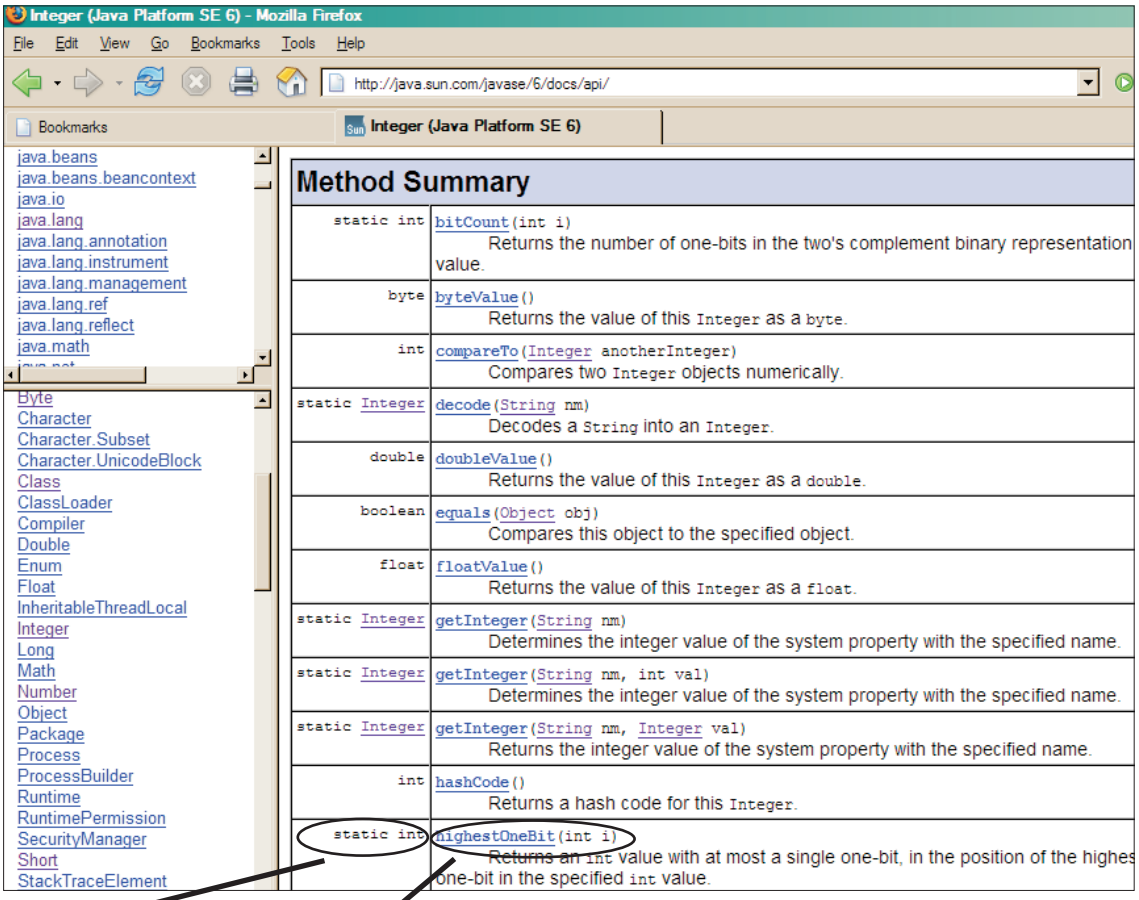
✗ Then this expression **<u>Displays:</u>**

```
System.out.println(work_int1.byteValue());      37
System.out.println(work_int2.byteValue());      126
System.out.println(work_int3.byteValue());      65

System.out.println(work_int1.doubleValue());    37.0
System.out.println(work_int2.doubleValue());    23678.0
System.out.println(work_int3.doubleValue());    -1215.0

System.out.println(work_int1.floatValue());     37.0
System.out.println(work_int2.floatValue());     23678.0
System.out.println(work_int3.floatValue());     -1215.0

System.out.println(work_int1.longValue());      37
System.out.println(work_int2.longValue());      23678
System.out.println(work_int3.longValue());      -1215

System.out.println(work_int1.shortValue());     37
System.out.println(work_int2.shortValue());     23678
System.out.println(work_int3.shortValue());     -1215
```

# The Integer Wrapper Class, 5

- ❒ **The values for byteValue() of work_int2 and work_int3 are the result of the fact that the input values are out of the range of bytes (-128 to +127) - see course Appendix A on conversion rules for more details**

- ❒ **Important point here: how to read the documentation write ups for methods:**




- ♦ **Note the left hand column of the Method Summary specifies the type of value the method returns**

  - ✗ If the word **static** is present, the method applies to the class not instances, so invoke the method as **Integer.***method_name()*

- ♦ **Then the right hand column specifies the method name and the arguments the method expects, along with a short description**

# The Integer Wrapper Class, 6

❒ **Remember also that if you scroll down further on this page, you will find more detailed descriptions of fields and methods**

- ♦ **Sometimes not much more detailed, other times with some nice, richer, helpful information**

❒ **Continuing our examination of Integer methods ...**

- ♦ **Integer.bitCount(*int*) - returns the number of one bits in *int* (where *int* represents a primitive integer literal or variable)**
- ♦ **Integer.highestOneBit(*int*) - returns the leftmost one bit in *int* (as a power of two)**
- ♦ **Integer.lowestOneBit(*int*) - returns the rightmost one bit in *int* (as a power of two)**
  - ✗ For each of the two methods above, the value returned is '1' if the rightmost bit is the answer, '2' if the second rightmost bit is the answer, ... '-2147483648' if the leftmost bit is the answer
- ♦ **Integer.numberOfLeadingZeros(*int*) - returns the number leading zero bits in *int***
- ♦ **Integer.numberOfTrailingZeros(*int*) - returns the number of trailing zero bits in *int***
- ♦ **Integer.signum(*int*) - returns -1 if *int* is negative, 0 if *int* is 0, and +1 if *int* is positive**

# The Integer Wrapper Class, 7

❒ **Continuing our examination of Integer methods ...**

- **Integer.reverse(*int*) - returns *int* with order of the bits reversed**

- **Integer.reverseBytes(*int*) - returns *int* with order of the bytes reversed**

- **Integer.rotateLeft(*int*, *no_bits*) - returns *int* with bits rotated left the number of bits specified in the second integer argument *no_bits*; bits that rotate off the number to the left are rotated to appear on the right end**

- **Integer.rotateRight(*int*, *no_bits*) - returns *int* with bits rotated right the number of bits specified in the second integer argument *no_bits*; bits that rotate off the number to the right are rotated to appear on the left end**

- ***work_int.*hashCode() - an instance method that returns a hash code created from the current value of the instance (overriding the method of the same name in Object); assumes *work_int* is an Integer instance; for Integer instances, the hashCode returns the int value**

  ✗ The methods on this page might be useful for creating your encoding / encrypting code

# The Integer Wrapper Class, 8

❒ **Continuing our examination of Integer methods ...**

- **Integer.getInteger("*prop*") - returns an Integer object with the value of the system property named *prop*; if there is no property with that name, or if it is empty, a value of "null" is returned**

- **Integer.getInteger("*prop*", *default*) - returns an Integer object with the value of the system property named *prop*; if there is no property with that name, or if it is empty, an Integer object with the *default* integer value is returned**

- **Integer.getInteger("*prop*", *default_Int*) - returns an Integer object with the value of the system property named *prop*; if there is no property with that name, or if it is empty, the specified *default_Int* Integer object is returned**

  ✗ Note: there is no standard system property that returns an integer, so these only return a non-default value for system properties specified on the command line (see pages 124-126)

**Note**

- **Since these methods return Integer objects, you can apply Integer methods to the returned values, for example:**

```
System.out.println("The value of prop is " +
  (Integer.getInteger("prop")).intValue());
```

  ✗ Accesses the [user-defined] system property named prop, creates an Integer object with the value of prop, accesses that value, and displays it!

# The Integer Wrapper Class, 9

❒ **Continuing our examination of Integer methods ...**

- **Integer.toBinaryString(*int*) - returns a string of ones and zeros, with leading zeros omitted, that represents the value of *int***
- **Integer.toHexString(*int*) - returns a string of hex digits, with leading zeros omitted, that represents the value of *int***
- **Integer.toOctalString(*int*) - returns a string of octal digits, with leading zeros omitted, that represents the value of *int***
- **Integer.valueOf(*int*) - returns an Integer object with the value of *int***
- **Integer.valueOf(*String*) - returns an Integer object with the integer value of *String***
- **Integer.valueOf(*String*, *radix*) - returns an Integer object with the integer value of *String*, where *String* is passed as a base *radix* integer**

**Note**

✗ Testing demonstrates that for at least these methods one can pass an Integer object as the first argument and Java will do the appropriate conversion / extraction of values

# The Integer Wrapper Class, 10

❒ **Continuing our examination of Integer methods ...**

- *work_int*.**toString() - returns a String object with value the string of decimal digits that represent the value in the** *work_int* **object**

- **Integer.toString(***int***) - returns a String object with value the string of decimal digits that represent the value of** *int*

- **Integer.toString(***int***,** *radix***) - returns a String object with value the string of digits that represent the value of** *int***, where int is treated as a number specified in base** *radix*

- *work_int.***compareTo(***work_int2***) - compares two instances (or one instance to an integer value), returning 0 if they are equal, a negative value if the instance integer is less than the argument integer, or a positive number if the instance integer is greater than the argument integer**

- *work_int.***equals(***work_int2***) - returns a boolean** true **or** false

# The Integer Wrapper Class, 11

❒ **Continuing our examination of Integer methods ...**

- ♦ **Integer.decode(*string*) - returns an Integer instance with the integer value of a string; the string must be a valid octal, decimal, or hexadecimal integer or an exception occurs**

- ♦ **Integer.parseInt(*string*) - returns an integer value; string must contain a valid decimal integer or an exception occurs**

- ♦ **Integer.parseInt(*string*, *base*) - returns an integer value; *string* must contain a valid integer in the specified base (*base* itself is an integer value) or an exception occurs**

❒ **Several methods have mentioned that it's possible for an exception to occur - we shall examine exceptions in the next section**

# Other Wrapper Classes

❒ **As we mentioned at the beginning of this section, Java provides other wrapper classes for its other primitive data types; the list again:**

| Primitive type | Wrapper class |
|---|---|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

❒ **Now that you've seen what documentation is available for the Integer class, and how to read it, you can probably manage the other wrapper classes, no problem**

- ♦ **Basically, wrapper classes provide methods for conversion, value extracting, and testing / comparing in a formal way**

# Boxing and Unboxing

❒ **When a comparison or other operation requires it, the JVM will automatically convert the value of a primitive type to the corresponding value of its wrapper class (*e.g.*, int -> Integer)**

- ♦ **This automatic wrapping of a primitive is called** boxing**, or a** boxing conversion
- ♦ **The reverse operation is called** unboxing**, or an** unboxing conversion

❒ **Boxing requires allocating storage to hold the wrapper instance and then initializing the object's value to that of the primitive**

- ♦ **Then, depending on the operation, the appropriate method is invoked on new instance**

**Example - based on lab "Parser"**

```
int[] odds = {1, 3, 5};
int workInt;
System.out.println("odds[1] + 5 = " + odds[1]+5);
workInt = odds[1] + 5;
System.out.println("odds[1] + 5 = " + workInt);
```

**output:** **odds[1] + 5 = 15** **<-- boxed to Integer, converted toString, then concatenation**

**odds[1] + 5 = 6** **<-- calculation done first, then converted to String**

❒ **These operations can cause performance problems, so it is best to avoid them if at all possible**

## Computer Exercise: Wrapper Classes

In your J510 directory is a file named Wrapper_int.java. This file contains tests of all the methods in the Integer class; compile it and run it. You should actually run it several ways:

* simple execution:
  java Wrapper_int

* passing an argument (integer string):
  java Wrapper_int 27

* pass a system parameter named prop, with an integer value:
  java -Dprop=43 Wrapper_int

* pass both a system parameter named prop and an integer argument:
  java -Dprop=45 Wrapper_int 39

Finally, run this code so it will fail with an exception:
  java Wrapper_int abc

Feel free to experiment with the given code, as time and curiousity permit.

Also: starting with the page for the Integer wrapper class (see page 197), explore, briefly, the information available by clicking on all the links on the top three lines. Check out the other wrapper class descriptions mentioned two pages ago.

# Section Preview

- **Exceptions**
  - **Exception concepts**
  - **Exception handling**
  - **Selected Exception classes**
  - **The Throwable class and its methods**
  - **The implication of the exception hierarchy**
  - **finally Blocks**
  - **Defining your own exceptions**
  - **The try, catch, finally, and throw Statements**
  - **The throws clause**
  - **Chained exceptions**
  - **Exception Handling (Machine Exercise)**

# Exception Concepts

- **The developers of Java were very keen to create a language and runtime environment that was robust:**
  - **Include the ability to detect errors at compile time and run time**
  - **Provide facilities to handle or recover from errors where possible**
  - **Enforce the inclusion of error handling routines where practicable**

- **An exception is an event that signals a change to the flow of execution in your program**
  - **Most exceptions are errors that you may want to try and handle**
  - **However, some exceptions may indicate normal, expected events that are simply not predictable when they will occur**
    - ✗ For example, reaching end of file when reading a flat file

# Exception Concepts, 2

- **When you compile a Java program, the compiler checks for exceptions it knows are possible**
  - **And creates a compile time error if the code does not include error handling for those errors**
  - **These errors are called checked exceptions**
- **There are also errors that might only be detectable at run time**
  - **The unchecked exceptions classes are**
    - ✗ The class **RuntimeExceptions** and its subclasses and
    - ✗ The class **Error** and its subclasses

# Exception Concepts, 3

❒ **The application developer is not expected to handle unchecked exceptions**

- ♦ **But sometimes you can use handling of unchecked exceptions as a way to test values and recover from some errors**

❒ **You are required to acknowledge the possibility of checked exceptions in one of two ways:**

- ♦ **In your method header, if a checked exception can occur in that method, include a "throws" clause that lists all the checked exceptions that might occur and that your method will <u>not</u> be handling**

  ✗ You may include unchecked exceptions in the list if you wish, but the compiler only cares about checked exceptions

- ♦ **In your method body, enclose statements that may cause exceptions to be raised in a** try **block, and include** catch **blocks for handling specific exceptions, and possibly a** finally **block**

# Exception Concepts, 4

❒ **To understand exception handling these are the major concepts to grasp firmly:**

- ♦ **Each exception is a class, with fields and methods**
- ♦ **All exceptions are direct or indirect children of the Throwable class (and so inherit its fields and methods)**
- ♦ **When an exceptional condition is encountered, an exception is created (an instance of that exception class is instantiated using** new **and invoking a constructor) and passed to the runtime using the** throw **statement**

**For example**

```
if (locLang.equals("NF")) {
     throw new LanguageGoneException()
   }
```

❒ **Usually, Java-provided exceptions are thrown by the system on your behalf**

- ♦ **But you can define your own exception classes and throw them yourself - the above example shows throwing a user-defined exception**

❒ **Appendix B to this book includes a listing of all Java-provided exceptions as of JDK 6**

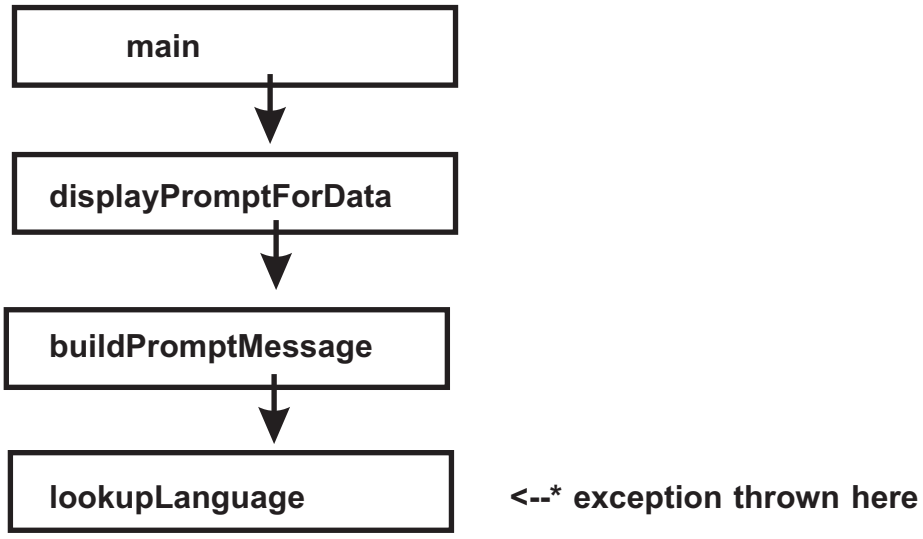
# Exception Concepts, 5

❒ **When an exception is thrown, the run time goes on a search for a routine that has volunteered to catch this particular exception**

- ♦ **The search begins at the method that was running at the time the exception was thrown and progresses up the call chain until the exception is caught**

  - ✗ If no routine catches the exception, the thread is terminated

❒ **For example, every application always runs from the main method, which then invokes other methods, which may invoke other methods ... so we might see something like this**




- ♦ **At this point, the runtime works back up the stack, looking for some routine (**catch **block or** finally **block in the stack of method calls) to handle the particular exception thrown**

# Exception Concepts, 6

❒ **Although a class may include many methods, only the methods in the direct call chain are examined for exception handlers**

- ♦ **An exception handler is a** catch **block** that **names an exception explicitly**
- ♦ **It must be related to a** try **block**

❒ **The structure, then, is something like this:**

```
try {
      - statement(s) that may raise exception(s)
    }
catch (exception_name variable_name) {
      - statement(s) to handle named exception
      }
catch (exception_name variable_name) {
      - statement(s) to handle named exception
      }
.
.
.
catch (exception_name variable_name) {
      - statement(s) to handle named exception
      }
finally {
      - statement(s) to run after try block exits
          [this block is run whether or not
           a catch block is run]
      }
```

**<== use the** *variable_name* **in the** catch **block to reference the exception object.**

# Exception Concepts, 7

❒ **Place statements that may raise exceptions inside** try **blocks - otherwise the** catch **blocks are not allowed**

- ♦ **You may place each statement in a** try **block followed by its own** catch **block**
- ♦ **Or a** try **block may contain several statements that, among them, might raise several exceptions**
  - ✗ Then follow with a series of **catch** blocks for all the possible exceptions

❒ **Note that for a** catch **block to be associated with a** try **block, there must be no code between the end of the** try **block and the start of the** catch **block series, nor any code between** catch **blocks**

- ♦ **Similarly, if there is a** finally **block, it must follow any** catch **blocks after a** try **block, with no intervening code**

❒ **Neither** catch **blocks nor** finally **blocks can exist outside the context of some** try **block**

❒ **Note also that constructors and initializers can throw exceptions**

- ♦ **And they may have** try**,** catch**, and** finally **blocks, also**
  - ✗ Constructor declarations may have a throws clause, and an initializer declaration may have a throws clause under some circumstances too complex to discuss in this course

# Exception Handling

❒ **So what can you do in an exception handler (**catch **block)?**

- ♦ **Just about anything, including**
  - ✗ Updating fields
  - ✗ Invoking methods, including methods of the current exception class, or its parent or ancestor methods
  - ✗ Emitting messages
  - ✗ Logging or recording related information for later examination
  - ✗ Requesting user direction
  - ✗ Throwing another exception
  - ✗ Issuing **return**
- ♦ **Once the code in an exception handler is run, control returns to the point after where the exception occured, and execution continues**
  - ✗ Unless you issued **return**, in which case the program ends
    - ➢ Reminder: **return** cannot be issued in a constructor nor in an initializer

# Selected Exception Classes

❒ **We will show some examples of code for working with the most common exceptions we encounter in this course**

**Checked exception classes of interest**

```
Object
    Throwable
        IOException
            EOFException
            FileNotFoundException
```

**Unchecked exception classes of interest**

```
Object
    Throwable
        Exception
            RuntimeException
                BufferOverflowException
                BufferUnderflowException
                NumberFormatException
                IndexOutOfBoundsException
                    ArrayIndexOutOfBoundsException
                    StringIndexOutOfBoundsException
```

❒ **The indentation shows the subclassing structure, which, we will soon find, is important in some coding decisions**

- **This seems to be a good place to point out that the name of an exception class usually (not always) ends in "Exception"**

# The Throwable Class and its Methods

❒ **Because they may be useful in an exception handling routine, you should be aware of the methods available to you for each particular exception**

- **And since all exception classes have Throwable as a direct or indirect parent, these classes will generally inherit Throwable's methods**

**Methods in class Throwable (all are instance methods)**

- **getCause() - returns the throwable that caused this exception to be thrown, or** null
- **getMessage() - returns the message string associated with this exception or** null
- **getLocalizedMessage() - returns a locale-specific version of the message string; unless overridden by a subclass, returns the same as getMessage() or** null
- **toString() - returns a short description; if getLocalizedMessage() returns** null**, this method returns just the class name**
- **initCause(***cause***) - initializes a Throwable before being thrown; more of interest when we discuss chained exceptions later**

# The Throwable Class and its Methods, 2

- **printStackTrace() - writes the throwable and its backtrace to the System.err stream (usually stderr)**

- **printStackTrace(*printstream*) - same as printStackTrace(), but allows you to direct the output to some other output stream**

- **printStackTrace(*writer*) - same as printStackTrace(), but allows you to direct the output to a specific PrintWriter object**

  - ✗ Note: the above three methods return **void**, so they should be used as verbs

- **getStackTrace() - returns an array of stack trace elements containing the lines of output that appear in printStackTrace() calls**

  - ✗ One could process each element of this array using StackTraceElement methods, the most useful of which might be: **getClassName()**, **getFileName()**, **getLineNumber()**, **getMethodName()**, **isNativeMethod()** [returns true or false], and **toString()**

- **fillInStackTrace() - returns a reference to this Throwable instance; intended to be used internally to this Throwable's class**

- **setStackTrace(*StackTraceElement[]*) - passing an array of stack trace elements; used by "RPC frameworks and other advanced systems" to build a stack trace in an alternative way**

# The Throwable Class and its Methods, 3

- **Note: the printStackTrace() and getStackTrace() methods allow you to write to / access the stack trace, which is generally not done if an exception handler returns control to the program normally**

- **Usage for getStackTrace() might work like this:**

```
StackTraceElement[] workStackTrace;
```

  - **Then, in an exception handler, you might code something like this:**

```
workStackTrace    = ex.getStackTrace();
if (workStackTrace.length > 0)
   {
    for (int i = 0; i < workStackTrace.length; i++)
       {
        System.out.println("StackTraceElement["+i+"] = "
                    + workStackTrace[i]);
       }
    }
```

  - **In the inner loop you could also apply the various StackTraceElement methods against each element in the array**

- **The point here is that in an exception handler (**catch **block) there is a lot of information available to help you solve, recover from, or otherwise handle the exception**

# The Implication of the Exception Hierarchy

❒ **Here, let's put together two pieces of information we've already seen:**

**The general structure of exception handling in a method:**

```
try {
        - statement(s) that may raise exception(s)
    }
catch (exception_name variable_name) {
        - statement(s) to handle named exception
        }
catch (exception_name variable_name) {
        - statement(s) to handle named exception
        }
.
.
.
catch (exception_name variable_name) {
        - statement(s) to handle named exception
        }
finally {
        - statement(s) to run after try block exits
                [this block is run whether or not
                 a catch block is run]
        }
```

**Some specific exception classes:**

```
Object
        Throwable
                IOException
                        EOFException
                        FileNotFoundException
```

# The Implication of the Exception Hierarchy, 2

- **Now, when an exception is thrown, all the related** catch **blocks are examined in the order they appear in the program**
  - **The first** catch **block whose** *exception_name* **matches the current exception will be run and program contol will resume after the last** catch **block in the list (in case there is a** finally **block)**
- **So if you were planning on handling, say, IOException, EOFException, and FileNotFoundException, if you put a** catch **block for IOException first, then the other two** catch **blocks would not be reached, since the IOException superclasses the other two**
  - **In fact, you would get an error message from the compiler**
- **But you can specify** catch **blocks for the subclass exceptions first, and then a** catch **block for the superclass**
  - **The ultimate in this would be to specify**

```
catch(Exception ex) {
   ex.printStackTrace();
   }
```

  - **This would** catch **all exceptions (if placed first in your sequence of** catch **blocks)**
    - ✗ Or all uncaught exceptions (if placed later in the list) - in which case any following **catch** blocks would never be entered
  - **The above** catch **block is not recommended as it basically bypasses all exceptions and keeps on running**

# finally Blocks

- **If you have a** finally **block, it is associated with a** try **block and appears after any and all** catch **blocks associated with that** try **block**

- **A** finally **block is always executed after execution of the** try **block**
  - **Whether or not an exception has been raised**
  - **Even if a return is issued in an exception handler**

- **A** finally **block might be a good way to ensure you do some clean up and orderly termination**
  - **But you may want to set some kind of boolean to make sure you only run the code under particular circumstance(s)**

# Defining Your Own Exceptions

❒ **It's hard to imagine a practical reason for creating your own exception classes:**

- ♦ **There is already a rich set of exception classes available**
- ♦ **It is almost always better to check for the exceptional condition and handle it inline or by simply invoking a repair method**

❒ **But it is possible and so we have put together an example using our case study:**

- ♦ **Suppose in some method we will be using an object variable:**

```
InventoryItem workItem;
```

  - ✗ This is an object reference variable and can hold a reference to any InventoryItem instance or any subclass instance
    - ➢ In particular, it could hold a reference to a Foodstuff instance

- ♦ **Further suppose in our logic we want to report on the expiration date for the item referenced in workItem, using getExpDate**
  - ✗ This is fine if workItem holds a Foodstuffs instance, but an error if workItem holds an InventoryItem instance, since only Foodstuffs instances have expiration dates

# Defining Your Own Exceptions, 2

❒ **So the pieces we have to set up are:**

- ♦ **Need to define a new exception class, let's call it WrongKindException**
- ♦ **Need to add methods to InventoryItem: getExpDate() and setExpDate(String)**
  - ✗ These mimic the signatures of the same methods in Foodstuffs
  - ✗ We have to include a throws clause and a throw statement in each
- ♦ **Finally, in our code that uses the workItem variable, we need to put the attempt to getExpDate in a** try **block and follow that with a** catch **block for our exception**

❒ **Now to implement all this ...**

# Defining Your Own Exceptions, 3

❒ **A user-defined exception needs to extend an existing exception**

- ♦ **Usually you start with the Exception class as the parent for your application-specific exception classes**
  - ✗ Later, you may find value in subclassing your exception class

❒ **We define an exception like any class, perhaps something like:**

```
public class WrongKindException extends Exception {
```

- ♦ **Strangely enough, we don't need any initializers, constructors or methods - methods will be inherited, all else is optional**
- ♦ **However, maybe we should put a little "breadcrumbs" method, just for show, so our whole exception class might look like this:**

```
package inventory;

// Definition of WrongKindException
// - a user-defined exception class

 public class WrongKindException extends Exception {

// Definition of method

    public void reportWrongKind() {
      System.out.println("In reportWrongKind");
    }

 }   // end of WrongKindException
```

# Defining Your Own Exceptions, 4

❒ **Next, we need to add the methods in InventoryItem that will throw this exception, maybe something like this (we just show one here):**

```
   .
   .
   .
// Definition of method getExpDate
    public String getExpDate()
                              throws WrongKindException
    {
       throw new WrongKindException();
    }
```

♦ **Notice that even though the method is defined as returning a String, you cannot include a** return **statement:**

✗ If the **return** appears before the **throw**, the compiler is unhappy that the **throw** is unnreachable

✗ If the **return** appears after the **throw**, the compiler is unhappy that the **return** is unnreachable

# Defining Your Own Exceptions, 5

- Finally, in our code that uses our generic InventoryItem object, we might have this

```
.
.
.
InventoryItem workItem;

InventoryItem class_up = new
      InventoryItem("PART1002","Initial item",2,7);

Foodstuffs     food_up = new
      Foodstuffs("PART5003", "Initial food", 3,
                 8.50f, "Sept. 2007");

try {
workItem = food_up;
System.out.println("Part number = " +
     workItem.getPno());

System.out.println("Expiration date = " +
     workItem.getExpDate());

workItem = class_up;
System.out.println("Part number = " +
     workItem.getPno());

System.out.println("Expiration date = " +
      workItem.getExpDate());
    }

catch (WrongKindException ex)
    { ex.reportWrongKind(); }
```

# Defining Your Own Exceptions, 6

❒ **And when we run the code, we see:**

```
Part number = PART5003
Expiration date = Sept. 2007
Part number = PART1002
In reportWrongKind
```

- ♦ **So we did, indeed get to our personal exception class!**

# The try, catch, finally, and throw Statements

❒ **To provide a formal description, we include all four statements here**

**Syntax**

**try {** *statement(s)* **}**

**catch (***exception_name variable_name* **) {** *statement(s)* **}**

**finally {** *statement(s)* **}**

**throw** *throwable_object()***;**

♦ *throwable_object* **is an object variable that has been initialized as an object that extends Throwable**

✗ Most often **new *exception_name()***

♦ **Note also that the** *exception_name* **can use a constructor other than the empty constructor**

✗ Perusal of the Throwable class description in the API docs shows you can also specify a constructor of a String (which is some message describing the error), another exception (in the case of chained exceptions, discussed shortly) or both

# The throws Clause

❒ **Method, constructor, and initializer headers may each contain a** throws **clause**

- ♦ **In all cases, the object of the clause is a comma-separated list of exceptions that might be raised in the related code**

  - ✗ These are normally checked exceptions, but the list may contain any list of exceptions

    **... throws** *exception***[,** *exception* **[...]];**

❒ **Note that when an exception is thrown, the current block of code (method, constructor, initializer) is said to complete abruptly**

- ♦ **Perhaps leaving unfinished business behind (instances not created, calculations not done, files not closed,** *etc.***)**

# Chained Exceptions

- **It is not uncommon for an exception handler to itself throw an exception**
  - **Perhaps one catch block can handle the current exception partially and then invoke a different exception to handle the remaining work**
    - ✗ This allows exceptions to be very specialiazed and to work together to solve complex exception problems

**For example**

```
try { ... code ...}
catch (BadMessageException ex)
   { ... code ...
     throw new LanguageException("Language missing", ex)
   }
```

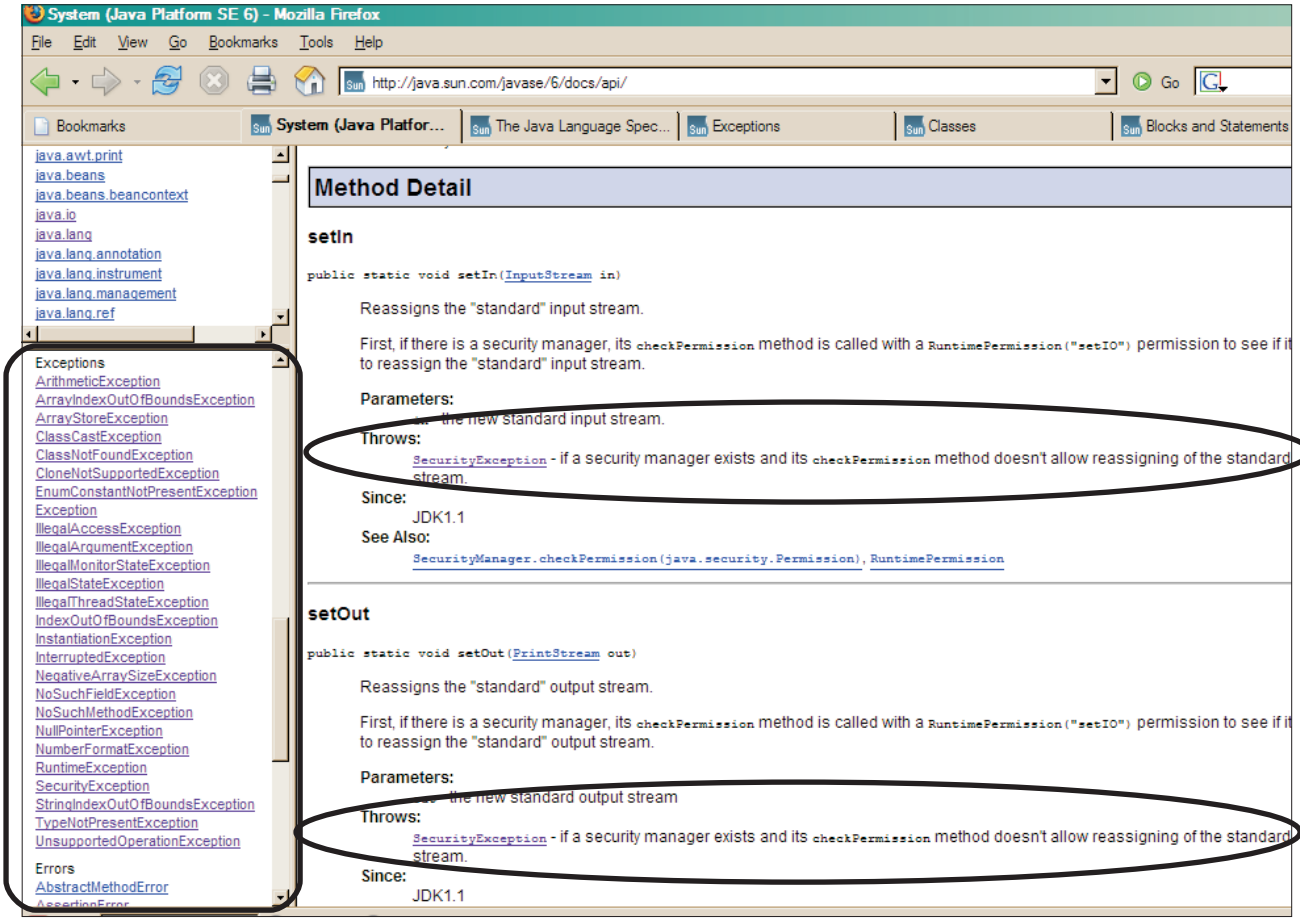
- **In this example, the LanguageException class is constructed using a message and a "throwable cause" (fancy way of saying we pass the cause of a previous exception on the exception chain along)**
- **This allows the LanguageException handler to understand what was the cause of the current failure, and what exception originally raised the current exception)**
- **Thus we see better the application of the getCause() and initCause() methods we mentioned earlier**

# More Information on Exceptions

❒ **The API docs contain helpful information regarding exceptions:**

- **In the classes panel, when you have selected a package, scroll down to find the exceptions and errors that might be thrown in each class**

- **In the details panel, scroll down and you'll see each method write up includes the exceptions that it might throw:**



## Computer Exercise: Exception Handling

Modify the program Wrapper_int.java from the last lab to add an exception handler. The pieces:

* Put a **try** block around the two lines 21-22 where the decode method is invoked

* Follow the **try** block with a **catch** block, for the exception named NumberFormatException; in this block:

    * write out some error messages
    * put a value of 5 in the work_in variable
    * invoke some of the Throwable methods, displaying the results; for example:

    System.out.println("getMessage() yields " + ex.getMessage());

    see pages 221-223 for the list of possible methods to try

* Compile the resulting code

* Test the results, deliberately introducing a number format exception, as on the previous lab:

    java Wrapper_int abc

This page intentionally left almost blank.

# Section Preview

- **Case Study Part 3**
  - **Exploring Input and Output**
  - **The System class**
  - **I/O Related Terms**
  - **Processing Streams**
  - **I/O Processing Exceptions**
  - **Command Line Reading**
  - **Skeleton Menu Logic**
  - **The InventoryDataManager class (Machine Exercise)**

# Exploring Input and Output

- In all our work so far, we have used System.out.println() to display data at the terminal
  - And input to our methods has been limited to system properties and command line arguments
- Now we begin our journey towards doing real input and output
  - However, we begin with a fuller exploration of the System class, which will lead us in a natural way to the whole I/O world

# The System Class

- **By now, you probably take System.out.println() for granted**
  - **But if you look at the API doc for the System class you find:**

- **System is a "final" class: it cannot be instantiated (makes sense, acutally)**

- **The System class has three fields,** in**,** out**, and** err
  - **The fields are not like fields we've been used to so far: these fields are instantiations of the InputStream and OutputStream classes**
    - ✗ **System.err** is the standard error stream - usually mapped to the terminal display, but it can be remapped to any desired OutputStream
    - ✗ **System.in** is the standard input stream - usually mapped to the keyboard, but it can be remapped to any desired InputStream
    - ✗ **System.out** is the standard output stream - usually mapped to the terminal display, but it can be remapped to any desired OutputStream
  - **So, System.out.println() actually represents the println() method of the OutputStream class applied to the standard output stream**

# The System Class, 2

❒ **Before we delve into the various stream kinds of classes, let's explore the System class methods that may be of use in our work**

- **System.getProperty(*name*) - we actually saw this earlier (p. 126); it returns the String containing the value of the system property specified as *name***

- **System.setProperty(*name*, *value*) - sets the *value* of the system property *name*, and returns a String containing the previous value, if any, of that property**

- **System.getenv(*env_var*) - returns a String containing the value of the specified environment variable, *e.g.*: System.getenv("HOME")**

  ✗ The docs point out this might yield different results on different platforms, primarily because of different rules regarding case sensitivity

  ➢ For example, UNIX is case sensitive, while Windows is case insensitive

- **System.arraycopy(*src*, *strt*, *dest*, *target*, *no.*) - copies elements from array *src*, beginning at the *strt*-th element, to array *dest*, beginning at the *target*-th element, for *no.* elements**

**Example**

```
System.arraycopy(evens, 0, odds, 0, 4);
```

# Exploring Input and Output, continued

- **There are four main classes we care about initially, all part of the java.io package, and each is a direct child of Object**
  - **InputStream - primarily concerned with reading bytes**
  - **Reader - primarily concerned with reading characters**
  - **OutputStream - primarily concerned with writing bytes**
  - **Writer - primarily concerned with writing characters**
- **Notice the parallelism**
  - **The above four classes are abstract classes: they cannot be instantiated, they must be subclassed to be used**
  - **Each of these classes have subclasses that also mirror each other (where it makes sense)**
  - **So, for example, data written using methods from the DataOutputStream subclass of OutputStream can be read back using methods from the DataInputStream subclass of InputStream**
    - ✗ And in some cases, other combinations are possible
    - ✗ Over time, we'll explore the relevant details

# I/O Related Terms

❒ **Before we go any further we need to define / clarify terms we'll be using:**

- **byte - 8 bits of storage / data; as an integer value, anywhere from -128 to +127 or 0 to 255; however, may hold any data (for example, image files are composed of bytes whose numeric integer value are of no interest** *per se*)

- **character - 16 bits of storage / data (in rare cases, a character may need 32 bits)**

- **stream - a series of bytes or characters; at the lowest level, Java streams are read from or written to one byte or one character at a time (each such operation requiring an invocation of an I/O service from the underlying platform)**

**All I/O streams in Java are byte streams or character streams**

- **buffer - contiguous memory as a place to gather bytes or characters into blocks before writing, or as a place to extract bytes or characters, in order to minimize calls to I/O services from the underlying platform**

- **filter - to transform data in some fashion before writing or after reading; for example, extracting field values from an object or converting a String to bytes or characters (or back), translating characters to a different code page**

# I/O Related Terms, 2

- **file - a file in the sense of the underlying platform; for the z/OS case, this can be an MVS file or HFS file**

- **file descriptor - a resource used to access a file; platform dependent implementation; some platforms have a limit on the number of file descriptors that can be in use for a thread at any one time; usually a small integer (0, 1, ... )**

- **mark - to save a position in an input stream in order to return to it later; only one position may be saved at a time; you may specify a threshold beyond which the current mark will be discarded**

- **reset - reposition to the last marked position in an input stream**

- **flush - for output, to write out remaining data in a buffer, ensuring the data is written, and freeing the buffer space for reuse**

  - ✗ In most cases, the JVM has to rely on the underlying platform to ensure that data directed towards external storage is actually written

- **close - to terminate reading or writing a file or stream, and to free up any file descriptors tied to a file; usually a close issues an implicit flush**

- **blocking - on input, putting a thread into a wait state until data arrives, end of file is encountered, or an I/O error occurs**

# Processing Streams

❒ **A couple of pages ago we pointed out:**

**All I/O streams in Java are byte streams or character streams**

- **This is in contrast to traditional z/OS files which usually are record oriented and where records may contain data in character fields, packed decimal, or binary integer**

❒ **Note that streams can be tied to**

- **Standard system input, output and error streams (usually the keyboard and display)**
- **HFS files (equivalently to zFS or NFS files also)**
- **MVS files**
- **Transient streams such as HTTP requests, CICS transactions**
- **Memory areas / strings**
- **Pipes**
- **Sockets**
- **Machines or other devices, data sources and data sinks**

# Processing Streams, 2

❒ **The first step in coding for I/O is to choose what I/O classes to use, and that depends on the what kinds of data you will be processing**

- ♦ **There are over two dozen I/O classes to choose from; here's an overview and comparison of the character stream classes and their corresponding byte stream classes (indentation shows subclassing, not all classes are shown):**

| Character stream classes | Function | Byte stream classes |
|---|---|---|
| Reader | Abstract class for input | InputStream |
| BufferedReader | Buffers input, parses lines | |
| LineNumberReader | Keeps track of line numbers | |
| FilterReader | Abstract class | |
| PushBackReader | Allow char(s) to be pushed back | |
| InputStreamReader | Translate byte stream to char stream | |
| FileReader | Translate bytes to characters | FileInputStream |
| | Transforms data | FilterInputStream |
| | Buffers input, parses lines | BufferedInputStream |
| | Reads Java primitive data types | DataInputStream |
| | Keeps track of line numbers | LineNumberInputStream |
| | Allows byte(s) to be pushed back | PushBackInputStream |

- ♦ **The correspondences do not always line up because we wanted to preserve the subclassing information in both columns**

| Character stream classes | Function | Byte stream classes |
|---|---|---|
| Writer | Abstract class for output | OutputStream |
| BufferedWriter | Buffers output | |
| FilterWriter | <- abstract \| non-abstract -> | FilterOutputStream |
| | buffers output | BufferedOutputStream |
| | writes Java primitives | DataOutputStream |
| | writes to a printer | PrintStream |
| OutputStreamWriter | Transforms data | |
| FileWriter | Write to file | FileOutputStream |
| PrintWriter | Write to a printer | |

# Processing Streams, 3

❒ **Here are some common practices / recommendations**

- **For displaying character data on the terminal / console:**

  ✗ Continue to use System.out.print and System.out.println as before - note that these do not throw I/O exceptions

  ➢ We'll discuss more formatting options later

- **For reading from the terminal keyboard:**

  ✗ Use the readLine() method from the BufferedReader subclass of the Reader class

  ➢ And possibly methods from the Scanner utility to extract tokens, discussed later

**Note**

- **Although there is a System.in.read() method, it returns a single byte in an int field, and for that byte to be converted to char for examination takes a number of intermediate steps**

  ✗ We find the approach we discuss in a few pages on Command Line Reading to be more practical

# Processing Streams, 4

❒ **More stream processing recommendations**

- ♦ **For wrting to DASD:**
  - ✗ Use OutputStream subclasses for raw byte data such as images, PDFs, audio, and so on
  - ✗ Use the DataOutputStream subclass of OutputStream to write Java primitive data types in a way that can be read from any JVM
  - ✗ Use Writer subclasses to write text data to DASD, especially to encode in various codepages
- ♦ **For reading from DASD:**
  - ✗ Use InputStream subclasses for raw byte data - although for particular data types there may be classes available for processing certain kinds of data
  - ✗ Use the DataInputStream subclass of InputStream to read data written using DataOutputStream methods
  - ✗ Use Reader subclasses to read text data writen with Writer subclasses

❒ **We'll cover more details later on the various classes relevant to our work here**

# Processing Streams, 5

- **Right now, we want to choose a class for reading text from the terminal (**stdin**); our best option turns out to be to use the InputStreamReader class "wrapped inside" the BufferedReader class**

  - **In your code, to set up an object to use for reading, you could write something like this:**

```
BufferedReader inReader = new
      BufferedReader(new InputStreamReader(System.in));
```

- **However, in one of the Java tutorials you'll find this code, which seems cleaner and more flexible:**

```
InputStreamReader rdr1 = null;
BufferedReader inReader = null;
.
.
.
rdr1 = new InputStreamReader(System.in);
inReader = new BufferedReader(rdr1);
```

  - **This process is analogous to allocation and open for z/OS: the** stdin **stream is now tied to the inReader object, so you may now invoke methods against this object**

# I/O Processing Exceptions

- **Just about every I/O operation can throw one or more exceptions, most commonly**
  - **IOException - can be caused by a variety of reasons, but usually a physical (mechanical) error**
  - **FileNotFoundException - even when creating a file, in which case it is not usually an error but it is always an exception**
- **So you need to embed most I/O requests in** try **blocks and provide at least** catch **blocks for these two exceptions**

# Command Line Reading

- **In the next section we will begin working with the useful I/O options**
  - **But, as a preliminary to that, we first want to work on building a new class for our Case Study**
    - ✗ A class that will interact with the user from the command line to drive creating and displaying inventory items
- **In the real world we would probably use some kind of web interaction (filling out HTML forms, using CGIs or JSPs, and so on)**
  - **But we haven't evolved the infrastructure enough to do this yet**
- **Our new class, InventoryDataManager will engage the user at the command line level, providing a menu of choices and letting the user decide what to do**
  - **To do this requires a prompt (we know how to use System.out.println for this already)**
  - **And a way to access the user's response - how to read data keyed in at the command line (this we talk about next)**
  - **And a loop to stay involved with they user until he or she is ready to stop (we also know how to do this already)**

# Command Line Reading, 2

- Here's some psuedo-code to sketch out the generic logic of interacting with the user at the command line:

```
while still working...
    clear choice and input variables
    Provide list of choices for user
    read choice from command line
    case for choice
          option_1 - action(s)
          option_2 - action(s)
          .
          .
          .
          option_exit - set loop control variable to false
          otherwise - issue message about choices
end of loop
clean up
exit
// other methods follow
```

- Now our task is to convert this much into real Java code, and it might look like this ...

# Skeleton Menu Logic

- **Setup, main loop, and display prompt menu:**

```
class InventoryDataManager {
   public static void main(String[] args) {

boolean             stillWorking = true;
int                 menuChoice = 0;
String              lineIn = "";

InputStreamReader   rdr1 = null;
BufferedReader      rdr2 = null;

   rdr1 = new InputStreamReader(System.in);
   rdr2 = new BufferedReader(rdr1);

   while (stillWorking)
        {
lineIn = null;
menuChoice = 0;
System.out.println("Please select a choice from the list.");
System.out.println("   * Create / add data   ");
System.out.println("     1. Save as HFS file");
System.out.println("     2. Save as MVS file");
System.out.println("   * Display data   ");
System.out.println("     3. Data in HFS file");
System.out.println("     4. Data as MVS file");
System.out.println("   * Update data   ");
System.out.println("     5. Data in HFS file");
System.out.println("     6. Data as MVS file");
System.out.println("   * Delete data   ");
System.out.println("     7. Data in HFS file");
System.out.println("     8. Data as MVS file");
System.out.println("     x. exit   ");
System.out.print("Enter your choice ==> ");
```

# Skeleton Menu Logic, 2

- **Read from command line, and exception handling ...**

```
        try {
              lineIn = rdr2.readLine();
            }

        catch (IOException ex
             {
System.out.println("In IOException for menu routine.");
             }

      try {
            menuChoice = Integer.parseInt(lineIn);
          }
      catch (NumberFormatException ex)
           {
            if (lineIn.equalsIgnoreCase("x"))
               {
                stillWorking = false;
                System.out.println("OK. Leaving now.");
                return;
               }
           }
```

- **Notice that we read into a String variable (lineIn) and we have to use the Integer.parseInt method to convert it to an integer value, since the switch statement does not work on String values**

  ✗ Then, in case the user has not entered a valid numeric string, we need to catch any NumberFormatExceptions that might arise

    ➢ And if we get one, we see if the user entered a letter x, in upper or lower case, using the String method equalsIgnoreCase, which we have not yet covered in lecture but the working of which should be plenty obvious

# Skeleton Menu Logic, 3

❒ **Selection of action based on user request:**

```
      switch(menuChoice)
          {
           case 1: {; break;}
           case 2: {; break;}
           case 3: {; break;}
           case 4: {; break;}
           case 5: {; break;}
           case 6: {; break;}
           case 7: {; break;}
           case 8: {; break;}
           default:
              {
               if (stillWorking)
                  {
System.out.println("Select a choice from the list.");
System.out.println("Press <Enter> to retry.");
                   try {
                         lineIn = rdr2.readLine();
                        }
                   catch (IOException ex)
                        {
System.out.println("IOException waiting for Enter");
                        }

                  }  // end of if (stillWorking)

              }  // end of default processing

          }  // end of switch (menuChoice)

        }  // end of while (stillWorking)

      }  // end of main method

   }  // end of class
```

## Computer Exercise: The InventoryDataManager Class

In your J510 directory is a file named InventoryDataManager.java. This file contains code based on the code on pages 254 -256. Compile it.

Run InventoryDataManager.java. Test by entering the following values at the prompt:

1 - should re-display the menu
2 - should re-display the menu
e - should issue error message and re-display the menu
X - should end the session; then re-run and test:
x - should end the session.

This page intentionally left almost blank.

# Section Preview

- **Case Study Part 4**
  - **More on wrapper classes**
  - **Prompting and retrieving data from the command line**
  - **Some Application Design Issues**
  - **The DataManager class**
  - **Using the DataManager class**
  - **Using the DataManager class (Machine Exercise)**

# More on Wrapper Classes

- Earlier we explored the classes that Java supplies to provide methods to work with data primitives
  - We used the Integer class as the prime example, and covered it pretty thoroughly
- Here, we want to just highlight a number of methods that will prove useful in our work of getting data from the command line into an external data store
  - Since command line data, by definition, comes in as String data

# More on Wrapper Classes, 2

❒ **Methods to convert String to various formats (all are class methods)**

- ♦ **Byte.decode(*string*) - returns a Byte object from a String**
- ♦ **Byte.parseByte(*string*) - returns a signed decimal byte from a String**
- ♦ **Short.decode(*string*) - returns a Short object from a String**
- ♦ **Short.parseShort(*string*) - returns a signed short from a String**
- ♦ **Integer.decode(*string*) - returns an Integer object from a String**
- ♦ **Integer.parseInteger(*string*) - returns a signed int from a String**
- ♦ **Long.decode(*string*) - returns a Long object from a String**
- ♦ **Long.parseLong(*string*) - returns a signed long from a String**

  ✗ All these throw NumberFormatException if a bad value is found (not numeric or numeric but out of range)

# More on Wrapper Classes, 3

- **Float.parseFloat(*string*) - returns a float from a String**
  - ✗ Throws NumberFormatException if bad value (not numeric or numeric but out of range); note there is no Float.decode method
- **Double.parseDouble(*string*) - returns a double from a String**
  - ✗ Throws NumberFormatException if bad value (not numeric or numeric but out of range); note there is no Double.decode method

❒ **For a sense of balance and completeness, note that all these classes (Byte, Short, Integer, Long, Float, and Double) have a method called toString**

# Prompting and Retrieving Data From the Command Line

❒ **The general steps are**

- ♦ **Ask the user what choice they want**
- ♦ **Issue a readLine() to get the data**
- ♦ **Use a parse method to convert to the desired format**
  - ✗ Be sure to put that in a **try** with a **catch** for NumberFormatException

❒ **Such a method would require two arguments:**

- ♦ **Text of the prompt**
- ♦ **An open BufferedReader object reference**
  - ✗ Prompt will go to System.out stream
  - ✗ Input stream to use readLine() needs to be instantiated

# Some Application Design Issues

- Now we come to some complex design decisions

- We need to provide some methods to:

  - Prompt for data values and retrieve them - to create or update an object

  - Write the contents of data items to a stream / file - save to a data store external to memory

  - Locate / populate an object given a key / search value - retrieve from data store external to memory

  - Build an array of objects in memory

  - Display values - to command line or to HTML page / file

- So, our options are

  - Create a new class with all the relevant methods

  - Add new methods to the current class (InventoryDataManager)

  - Add new methods to, say, the InventoryItem class

# Some Application Design Issues, 2

❒ **This isn't an Object Oriented Design class, and so it isn't the place to go in to all the various methodologies and issues**

- ♦ **But a little thought will uncover the major pros and cons of the various alternatives ...**

**New class with the new methods**

- ✗ Pro: physically concentrate all these related methods in one place for ease of maintenance and packaging
- ✗ Con: invocation of methods will require a syntax of class_name.method_name(args) or even package_name.class_name.method_name(args)
- ✗ Pro: accessible from other classes that might not even be related to our current application - might be highly reusable

**New methods in InventoryDataManager**

- ✗ Pro: invocation syntax simpler: method_name(args)
- ✗ Pro: methods in class that uses them; easier to compile and make and test changes
- ✗ Con: not easily accessible if could be used by other classes

# Some Application Design Issues, 3

**New methods in InventoryItem class**

- ✗ Pro: methods close to class definitions
- ✗ Pro: methods inherited by Foodstuffs class
- ✗ Con: to be used by classes in other applications would require importing or extending this class, which would not be related - in other words, code reuse would be severly painful

❒ **To work in the Java style, it becomes clear that the right way to go is to build a new class, with only static methods, for prompting and retrieving different data items**

- ♦ **This encourages writing methods that can be used by other classes and, indeed, other applications**

❒ **The next question is: should the class that contains these new methods be stand-alone or should it extend an existing class (most likely InventoryDataManager itself)**

- ♦ **And, again, the decision ends in favor of creating a stand-alone class whose methods can be invoked from multiple applications**

# The DataManager Class

- So the first part of our work is to build a set of methods for prompting the user for information of various types, putting all these methods in a single, new class, DataManager
  - Perhaps like this (here are the front lines and the prompt for String data):

```
 // Class to handle general data get prompts
import java.io.*;
public class DataManager {

  // methods to prompt for fields of various types

 /* ********************************************** */


 public  static String PromptAndGetString(String text,
                                    BufferedReader in2)
    {
     String stringIn = null;

 System.out.println("Please enter the value you want for "
                   + text + ": ===> ");

        try {
              stringIn = in2.readLine();
             }
        catch (IOException ex)
             {
              System.out.println("In IOException for "+
                       "PromptAndGet string routine.");
              stringIn = null;
             }

 return stringIn;

    }  // end of prompt-and-get for string field
```

# The DataManager Class, 2

❒ The routine for working with integer data is, of course, very similar

```
/* ********************************************** */

public  static int     PromptAndGetInt(String text,
                                       BufferedReader in2
    {
     String stringIn = null;
     int backInt = -1;

System.out.println("Please enter the value you want for "
                    + text + ": ===> ");

       try {
             stringIn = in2.readLine();
             backInt = Integer.parseInt(stringIn);
            }

       catch (IOException ex)
            {
  System.out.println("In IOException for " +
                      "PromptAndGet int routine.");
            }

       catch (NumberFormatException ex)
            {
  System.out.println("In NumberFormatException for " +
                    "PromptAndGet float routine.");
            }

return backInt;

    }  // end of prompt-and-get for int field
```

- ♦ Notice how we set backInt to have a default value of -1, the traditional way to indicate error / failure

# The DataManager Class, 3

- Since we currently only have String, int, and float fields, we wrap up the DataManager class with the routine for float data:

```
/* ********************************************** */

public  static float  PromptAndGetInt(String text,
                                BufferedReader in2
    {
     String stringIn = null;
     float backFloat = -1f;

System.out.println("Please enter the value you want for "
                    + text + ": ===> ");

       try {
             stringIn = in2.readLine();
             backFloat = Float.parseFloat(stringIn);
            }

       catch (IOException ex)
            {
  System.out.println("In IOException for " +
                     "PromptAndGet float routine.");
            }

       catch (NumberFormatException ex)
            {
  System.out.println("In NumberFormatException for " +
                    "PromptAndGet float routine.");
            }

return backFloat;

    }  // end of prompt-and-get for float field

 }  // end of class
```

# Using the DataManager Class

- **Now assuming we get a clean compile, we can return to our work in the InventoryDataManager class menu work**
  - **We need to add a method that will invoke the DataManager class**

- **Recall that our menu choices from InventoryDataManager are:**

*** Create / add data**
**1. Save as HFS file**
**2. Save as MVS file**
*** Display data**
**3. Data in HFS file**
**4. Data in MVS file**
*** Update data**
**5. Data in HFS file**
**6. Data in MVS file**
*** Delete data**
**7. Data in HFS file**
**8. Data in MVS file**

- **We can use our PromptAndGet routines in the processing for options 1, and 2, and possibly in 5, and 6**
  - **So we decided to add a method in the InventoryDataManager class called getFields() that will prompt and get for all fields in our inventory objects: we can use this code from multiple locations**

# Using the DataManager Class, 2

❒ **So we add this method to our InventoryDataManager class:**

```
static void getFields() {

   workPartno = DataManager.PromptAndGetString(
                            "part number", rdr2);
   workDesc   = DataManager.PromptAndGetString(
                            "description", rdr2);
   workQOH    = DataManager.PromptAndGetInt(
                "initial quantity on hand", rdr2);
   workUnPr   = DataManager.PromptAndGetFloat(
                             "Unit Price", rdr2);

    }    // end of getFields method
```

**Notes**

- **We made the method "static" because it does not relate to any instance: it is a class method**
- **We used fields workPartno, workDesc, workQOH, and workUnPr so that we could share these values among all the methods in this class**
  - ✗ Which means we have to define them as "static", outside all the methods
- **Since we are passing the BufferedReader reference, we will have to define rdr2 as static, outside all the methods, also**

# Using the DataManager Class, 3

❒ **So the net of that is that the top part of InventoryDataManager.java now looks something like this:**

```
    class InventoryDataManager {

    static String          workPartno;
    static String          workDesc;
    static int             workQOH;
    static float           workUnPr;
    static BufferedReader  rdr2 = null;

       public static void main(String[] args) {

    boolean                stillWorking = true;
    int                    menuChoice = 0;
    String                 lineIn = "";

    InputStreamReader      rdr1 = null;
```

❒ **And, for starters, our menu processing logic might look like this:**

```
. . .
switch(menuChoice
    {
     case 1: {
                getFields();
      System.out.println("part number = " + workPartno);
      System.out.println("description = " + workDesc);
      System.out.println("initial QOH = " + workQOH);
      System.out.println("unit price  = " + workUnPr);
                ; break;}
. . .
```

♦ **The** println **commands, of course, are just temporary, to verify we got the data we were after; later we'll replace them with commands to write to HFS or MVS files, or to create objects**

## Computer Exercise: Using the DataManager Class

In your J510 directory is a file named DataManager.java. This file contains code based on the code on pages 267 -269. Compile this program.

Next, modify InventoryDataManager.java:

* Add the method getFields, as described on page 271; this belongs just before the end of class closing brace

* Move the declare of rdr2 to above the header for the main method, and declare it as static

* Add declares for four static fields, workPartno, workDesc, workQOH, and workUnPr, as shown on the facing page

* Add code for your case 1 logic as shown on the facing page, invoke getFields() then display the results using System.out.println

* Repeat this code under your case 2 logic

* Compile and run InventoryDataManager.java.

* Test options 1 and 2; make up your own data or you can use this:

| Part number | Description | QOH | Unit Price |
|---|---|---|---|
| PART00108 | Muddy Waters | 2 | 12.22 |
| PART00111 | Fudged Tests | 1 | 37.21 |
| PartyOn | Party kit | 50 | 49.99 |

This page intentionally left almost blank.

# Section Preview

- **Writing to HFS Files**
  - **File I/O in Java**
  - **The File class**
  - **Writing Fields**
  - **Classes for file output**
  - **OutputStream classes for file output**
  - **Using the FilterOutputStream class for file output**
  - **Using the BufferedOutputStream class for file output**
  - **Using the DataOutputStream class for file output**
  - **Using the PrintStream class for file output**
  - **Using the FileOutputStream class for file output**
  - **Writer classes for file output**
  - **Using the BufferedWriter class for file output**
  - **Using the OutputStreamWriter class for file output**
  - **Using the FileWriter class for file output**
  - **Using the PrintWriter class for file output**
  - **Summary and Comparison**
  - **Writing HFS Files from Java (Machine Exercise)**

# File I/O in Java

- **The z/OS concept of a "record" is really unknown in native Java**
  - **Data streams, if they go to external devices, are simply envisioned as streams**
  - **It's up to the programmer to interpret data**
- **So if we are to store our inventory items on disk, we're not really going to write them as records in a flat file**
  - **There's really two choices**
    - ✗ Keep the data for each object in a separate file - perhaps the filename would be a key for later retrieval; in our case, the part number (PartNo) would be a good choice for filename
    - ✗ Keep the data in an object or relational database - beyond the scope of this course
- **In preparing for this lecture, we created some test classes and we include here code fragments and results from various tests**
  - **The lab at the end of the section requires you to run these tests yourself - plus some more**

# The File Class

- **In the java.io package is a class called File**
  - **This class is an abstract representation of file and directory pathnames (but it is <u>not</u> an** abstract **class)**
- **Remember that Java is meant to be platform independent, so the File class provides mechanisms to handle common file-related tasks regardless of the underlying operating system**
  - **Here we give some details, but don't spend a lot of time working with the File class**
    - ✗ We cover it because File objects are used as inputs to some of the constructors for file I/O classes, and also we need at least one of the available methods later

# The File Class, 2

- **The File class has four fields, all static (class level)**
  - **separatorChar - the character used to separate names in a list**
  - **pathSeparatorChar - the character used to separate components in a path**
  - **separator - a String version of separatorChar**
  - **pathSeparator - a String version of pathSeparatorChar**
- **There are also four constructors available for instantiating a file**
  - **File(*pathname*) - given a pathname (possibly including a file name at the end), return a File instance**
  - **File(*parent*, *child*) - given a parent pathname and a child pathname, return a File instance with name constructed from the concatenation of the two (separated by the pathSeparator)**
  - **File(*file_instance*, *child*) - same as above, but the first operand is an already existing File instance**
  - **File(*URI*) - given a URI, return a new file instance**

# The File Class, 3

❒ **Finally, here are some of the methods available to the File class, just to give you some ideas of what's possible**

- ♦ **boolean instance methods: canExecute(), canRead(), canWrite(), delete(), exists(), isDirectory(), isFile(), isHidden(), makeDir(), setExecutable()**

- ♦ **String instance methods: getAbsolutePath(), getCannonicalPath(), getName(), getParent(), getPath(), toString()**

- ♦ **long instance methods: getTotalSpace(), getUseableSpace(), lastModified(), length()**

❒ **This is, as we said, only a partial list, but you get the idea of what's possible - pretty impressive!**

- ♦ **If and when you have a need to use these methods, refer to the API documentation for details**

  ✗ Note we do use the **exists()** method later (see page 371)

# The File Examples

- **We will be using our case study InventoryItem objects in our examples**
  - **Each output file will have a filename equal to the part number value:** *instance*.**getPno()**
  - **Each output file will have one record, consisting of the values found in the four fields**
    - ✗ The values used in the examples are short and simple
  - **In our examples, we will show the code used to write the file, then the resulting file viewed under** obrowse
- **Also note that while many of our examples will write the data to files in various character sets (codepages) the file names will always end up being EBCDIC, the native character set for z/OS**
  - **It is possible to have non-EBCDIC file names under the HFS, but they are very difficult to work with**

# Writing Fields

- **When writing out to a display, or a terminal, or even a printer:**
  - **Character data can be pretty free form**
    - ✗ In some cases, you might bound it by HTML tags
  - **Numeric data needs to be converted to character strings**
    - ✗ Perhaps with additional formatting, and also HTML tags
- **When writing to disk or tape with the need to read it back in some other time, fields within a record need to be fixed length so individual fields can be located**
  - **For character and String fields, we will write data out in fixed length chunks**
  - **For numeric fields, we try both native formats (primitives) and fixed length string formats, depending on what the various classes and methods allow**
- **To accomplish these feats will require we use some methods of the String class not yet discussed formally, including:**
  - **substring(*strt*[,*end*]) - operates on String instances to extract a substring, starting at offset *strt* through offset *end***
  - **length() - operates on String instances and returns an int containing the length of the instance**
  - **indexOf(*str*) - operates on String instances and returns an int containing the offset where the first occurrence of *str* is found**
  - **And a few others we introduce as we need them**

# Writing Fields, 2

❒ **Technique to write out fixed length String data: append field data with a string of blanks, then use the String substring() method to grab a subset of the size you need; for example:**

```
String workDesc;
String blanks = "                              ";

workDesc = item.getDesc() + blanks;
workDesc = workDesc.substring(0,30);
```

❒ **Technique to write out int data as fixed length String, when converting to String: use the Integer.toString method, prefix with a string of zeros, then use the String substring() method to grab the rightmost 9 characters**

- **An integer can hold numbers with up to nine decimal digits, so we prefix the converted int string with nine digits, then calculate the offset into the resulting string; for example:**

```
String workQOH;
int workOff;
String zeros = "000000000";

workQOH = zeros + Integer.toString(item.getQOH());
workOff = Integer.toString(item.getQOH()).length();
workQOH = workQOH.substring(workOff);
```

- **Note: when substring() does not have the second operand specified, it implies "to the end of the field"**

# Writing Fields, 3

- **Technique to write out** float **data as a fixed length String**
- **Note: when using the Float.toString(*float*) method, you need to know something about the range of the float data**
  - **If the value is between $10^{-3}$ and $10^{7}$ (.001 and 10,000,000), the returned string will be of the form** *dec_digits*.*dec_digits* **- no leading zeros nor trailing zeros, but there must be at least one digit on each side of the decimal point**
    - ✗ If the magnitude is in this range but the value is negative, there will be a leading minus sign ('-')
  - **If the value is outside this range, the string is in the form called "computerized scientific notation" - not discussed in this course**
- **For our application, we currently only have unitPrice as a floating point field, and it's range is hereby declared to be from 0.01 to 1,000,000.00, of course always positive, and always with exactly two decimal positions to the right of the decimal point**
- **So use the Float.toString method, prefix with a string of zeros, append one zero, then use the String indexOf(".") method to locate the decimal point**
  - **Finally, build the desired string as seven digits, a decimal point, and two digits; for example (**workOff **and** zeros **are as before):**

```
String workUnPr;
int workIndex;

workUnPr = zeros + Float.toString(item.getUnPr()) + "0";
workIndex = workUnPr.inexOf(".");
workOff = workIndex - 7;
workUnPr = workUnPr.substring(workOff,workIndex+3);
```

# Classes for File Output

❒ **When talking about streams that will write to HFS files, we have just these classes available:**

- **OutputStream - the abstract class that is the parent of all byte stream output classes**
  - ✗ FileOutputStream - a "convenience class" for writing byte files; extends OutputStream
  - ✗ FilterOutputStream - supports codepages; extends OutputStream
    - ➢ BufferedOutputStream - supports buffering; extends FilterOutputStream
    - ➢ DataOutputStream - supports writing Java primitives in a portable way; extends FilterOutputStream
    - ➢ PrintStream - supports print() methods; extends FilterOutputStream
- **Writer - the abstract class that is the parent of all character stream output classes**
  - ✗ BufferedWriter - supports buffering; extends Writer
  - ✗ OutputStreamWriter - bridge from character streams to byte streams; supports encoding; extends Writer
    - ➢ FileWriter - a "convenience class" for writing character files; extends OutputStreamWriter
  - ✗ PrintWriter - prints formatted representations of Strings and primitives; extends Writer

# Classes for File Output, 2

❒ **How do you choose, how do you use, these classes?**

- ♦ **On the following pages we discuss each of the nine non-abstract classes, including showing samples of code and the resulting output**
  - ✗ While extensive, it is not exhaustive, in that we haven't talked about all methods for all classes - but we provide a framework and some solid background for further exploration on your own

❒ **For the OutputStream classes, the constructors generally need a [first] argument of "an output stream"**

- ♦ **This is usually fulfilled by an instantiation of FileOutputStream:**

  **new FileOutputStream(***filename***)**

  **produces an output stream that will write to the file named** *filename*

❒ **For the Writer classes, the constructors generally need a [first] argument of "a writer"**

- ♦ **This is usually fulfilled by an instantiation of FileWriter:**

  **new FileWriter(***filename***)**

  **produces a Writer stream that will write to the file named** *filename*

# OutputStream Classes for File Output

- **We start by examining OutputStream classes**
- **And we begin with a short summary of the methods available in the parent (abstract) class, since all the subclasses, of course, inherit these methods**

**OutputStream methods**

- **close() - flush the stream then free the resources**
- **flush() - write remainder of any buffer**
- **write(*int*) - send the low order 8 bits of *int* to the stream as a byte**
- **write(*byte_array*) - write the byte array to the stream**
- **write(*byte_array*, *strt*, *len*) - write part of the byte array to the stream (begin at the *strt*-th entry, for *len* bytes)**

# Using The FilterOutputStream Class for File Output

❒ **The FilterOutputStream class extends OutputSteam and implements its methods**

**Constructor**

♦ **FilterOutputStream(***output_stream***)**

**Example**

```
InventoryItem testItem1 = new
 InventoryItem("PART00105", "Hogwash", 1, 12.50f);

filename = testItem1.getPno();
FilterOutputStream Rec1Out = new
 FilterOutputStream(new FileOutputStream(filename));
```

♦ **This establishes the filename, creates an OutputStream to write to that file, and instantiates a FilterOutputStream over that OutputStream**

✗ This FilterOutputStream is what we will use to write our first HFS files

# Using The FilterOutputStream Class for File Output, 2

- **Now, we need to write out our fields, of which two are Strings, one is int, and the other is float**
  - **But the only possibly relevant available method is write(*byte_array*)**
  - **So we need to convert each of our fields to byte arrays before we can write them**
  - **If we go to the String class API writeup, we find some help in these methods:**
    - *string*.**getBytes() - returns a byte array of this String instance, encoding the utf-16 characters into the platform's default character set**
    - *string*.**getBytes(***charset_instance***) - returns a byte array of this String instance, encoding the utf-16 characters into the character set represented in the** *charset_instance* **object**
    - *string*.**getBytes(***charset_name***) - returns a byte array of this String instance, encoding the utf-16 characters into the character set known as** *charset_name*
  - **You can find the complete list of official character set names on the internet at http://www.iana.org/assignments/character-sets**
  - **For our purposes we will work with:**

    **ISO-8859-1 - also known as "Latin1" and "ASCII"**
    **CP037 - also known as "Ebcdic"**
    **UTF-16 - also known as "ISO-10646" (Unicode)**

    - Note that these names are case-insenstitive, and utf-16 may also be written as utf16

# Using The FilterOutputStream Class for File Output, 3

- So, we can take our String fields, convert them to a byte array, in almost any codepage, and write them to an HFS file
  - And we take our int and float fields and first convert them to fixed length strings, as discussed earlier
  - The code we actually used in developing our example:

```
workPartNo = testItem1.getPno() + blanks;
workPartNo = workPartNo.substring(0,9);
Rec1Out.write(workPartNo.getBytes("ISO-8859-1"));

workDesc = testItem1.getDesc() + blanks;
workDesc = workDesc.substring(0,30);
Rec1Out.write(workDesc.getBytes("ISO-8859-1"));

workQOH = zeros + Integer.toString(testItem1.getQOH());
workOff = Integer.toString(testItem1.getQOH()).length();
workQOH = workQOH.substring(workOff);
Rec1Out.write(workQOH.getBytes("ISO-8859-1"));

workUnPr = zeros+Float.toString(testItem1.getUnPr())+"0";
workIndex = workUnPr.indexOf(".");
workOff = workIndex - 7;
workUnPr = workUnPr.substring(workOff,workIndex+3);
Rec1Out.write(workUnPr.getBytes("ISO-8859-1"));

Rec1Out.close();
```

**Note**

- Although you could mix code pages in one file, it would in reality be a very bad idea

# Using The FilterOutputStream Class for File Output, 4

- The code we've shown so far creates a file named PART00105 in the current directory
  - If we look at the file using obrowse, it looks like this:

```
& êè.....ç?ÅÏ/ËÇ.....................................
```

  - Not too surprising, given we have an ASCII variant here
  - Turning hex on shows:

```
& êè.....ç?ÅÏ/ËÇ.....................................
5455333334667676222222222222222222222333333333333333233
0124001058F771380000000000000000000000000010000012E50
```

  - BUT ... turning hex off and issuing: dis ascii **gives us:**

```
PART00105Hogwash                              000000010000012.50
```

- So here we have created a file in the HFS, encoded in ASCII!
  - What about other code pages?

# Using The FilterOutputStream Class for File Output, 5

❒ **We ran similar tests using code pages cp037 and utf16 (and slightly different data values), with these results:**

- **CP037 output:**

```
PART00106Logwash                                  0000000020000012.51
```

- **utf16 output (truncated here):**

```
Ú..&. .ê.è.........Ú..ë.?.Å.Ï./.Ë.Ç.....................
```

✗ After issuing **dis utf16 (still truncated)**:

```
.PART00107.Sogwash                                .00000003.0000012.52
```

✗ Notice the mysterious dots in front of each field; in hex we see:

```
.PART00107.Sogwash                                .00000003.0000012.52
FF0504050503030303FF0506060706070602020202020202020202
EF000102040000010007EF030F0707010308000000000000000000
```

✗ The leading dots are each x'FEFF' - this is a Unicode Byte Order Mark (BOM) which self-describes the data as being big endian

# Using The FilterOutputStream Class for File Output, 6

❒ **To put the process in context, we enclosed the write invocations in a** try **block with a** catch **block for IOException, something like this:**

```
try {
    filename = testItem1.getPno()
    FilterOutputStream Rec1Out =
        new FilterOutputStream(new FileOutputStream(filename));
    Rec1Out.write...
    . . .
    Rec1Out.close();

    filename = testItem1a.getPno()
    FilterOutputStream Rec1aOut =
        new FilterOutputStream(new FileOutputStream(filename));
    Rec1aOut.write...
    . . .
    Rec1aOut.close();

    filename = testItem1b.getPno()
    FilterOutputStream Rec1bOut =
        new FilterOutputStream(new FileOutputStream(filename));
    Rec1bOut.write...
    . . .
    Rec1bOut.close();

  }

catch (IOException ex) {;}
```

- **Actually, the constructors can't throw any exceptions so they could all be put outside the** try **block, but this approach seems to be sort of a "natural" organization of code**

# Using The BufferedOutputStream Class for File Output

- **The BufferedOutputStream class extends FilterOutputSteam and implements its methods**

  **Constructors**

  - **BufferedOutputStream(***output_stream***)**
  - **BufferedOutputStream(***output_stream***, *buf_size*)**

  **Example**

```
InventoryItem testItem2 = new
 InventoryItem("PART00205", "Hogwash", 1, 12.50f);

filename = testItem2.getPno();
BufferedOutputStream Rec2Out = new
 BufferedOutputStream(new FileOutputStream(filename),
     200);
```

  - **This establishes the filename, creates an OutputStream to write to that file, and instantiates a BufferedOutputStream over that OutputStream, with a 200 byte buffer (if *buf_size* < 0, throws an IllegalArgumentException)**

- **The choice of methods is the same as for the FilterOutputStream, and the results are equivalent**

  - **The only difference is that this stream is buffered, generally resulting in better performance (although this may only be apparent in long-running applications)**
  - **If you don't specify a buffer size, a default is selected for you**

# Using The DataOutputStream Class for File Output

❒ **The DataOutputStream class extends FilterOutputSteam and implements its methods**

**Constructor**

♦ **DataOutputStream(***output_stream***)**

**Example**

```
InventoryItem testItem2 = new
 InventoryItem("PART00205", "Hogwash", 1, 12.50f);

filename = testItem2.getPno();
DataOutputStream Rec2Out = new
 DataOutputStream(new FileOutputStream(filename));
```

❒ **But now we get something different ...**

# Using The DataOutputStream Class for File Output, 2

- **The methods available for the DataOutputStream class includes the methods inherited from the FilterOutputStream, but it also includes some new, interesting methods:**
  - **writeBoolean(*boolean*) - writes one byte**
  - **writeByte(*integer*) - writes one byte (righmost byte of *integer*)**
  - **writeBytes(*string*) - write out the rightmost byte of each character in the string**
  - **writeChar(*integer*) - writes rightmost two bytes**
  - **writeChars(*string*) - writes string of characters**
  - **writeDouble(*double*) - 8 bytes**
  - **writeFloat(*float*) - 4 bytes**
  - **writeInt(*integer*) - 4 bytes**
  - **writeLong(*long*) - 8 bytes**
  - **writeShort(*short*) - 2 bytes**
  - **writeUTF(*string*) - writes *string* in modified UTF-8 form, prefixed by a short int value containing the number of bytes in the output string**
- **Using these methods, character and string data are written out in utf16; numeric fields are converted to a standard internal format so they may be read back in from any a Java program on any platform**

# Using The DataOutputStream Class for File Output, 3

- The code we used in our testing was something like this:

```
RecOut.writeChars((testItem3.getPno()+blanks).
                  substring(0,9));
RecOut.writeChars((testItem3.getDesc()+blanks).
                  substring(0,30));
Rec2Out.writeInt(testItem3.getQOH());
Rec2Out.writeFloat(testItem3.getUnPr());
Rec2Out.close();
```

- Under obrowse, the output looks like this:

```
.&. .ê.è..........ç.?.Å.Ï.Ñ.Ë.Ç.............. ç..
```

  - **After** dis utf16**:**

```
PART00305Hogwish                          ....
```

  - **The dots on the right represent the int and float fields in their internal format; to see that in hex, we turn hex on:**

```
PART00305Hogwish                          ....
05040505030303030304060607060706020202020202020
00010204000003000508F0707090308000000000000000
```

- **Also note that the "internal format" for float and double, and so on, is specified in an international standard (IEEE-754), not just unique to Java, so other programs should also be able to interpret the data**

  - **... if you transfer the data in binary**

# Using The DataOutputStream Class for File Output, 3

❒ Just out of curiosity, and looking for patterns, we wrote out three files with slightly different int and float values

| Int value | 4 bytes of hex for int |
|---|---|
| 1 | 00 00 00 01 |
| 2 | 00 00 00 02 |
| 3 | 00 00 00 03 |

| float value | 4 bytes of hex for float |
|---|---|
| 12.50f | 41 48 00 00 |
| 12.51f | 41 48 28 F6 |
| 12.52f | 41 48 51 EC |

- ♦ Well, the integer is pretty clear, but the float would require more study than we care about in this course - we'll trust the compiler and the JVM to do what's right here

❒ Note there is no BOM even though the character portion is Unicode

❒ We can instantiate a DataOutputStream over a BufferedOutputStream, for example:

```
FileOutputStream out1 = null;
BufferedOutputStream bufout = null;
out1 = new FileOutputStream(filename);
bufout = new BufferedOutputStream(out1, 200);
DataOutputStream RecOuta = new DataOutputStream(bufout);
```

# Using The PrintStream Class for File Output

- **The PrintStream class extends FilterOutputSteam and implements its methods, plus some of its own**

  **Constructors**

  - **PrintStream(***File_object***)**
  - **PrintStream(***File_object***, *charset_name*)**
  - **PrintStream(***output_stream***)**
  - **PrintStream(***output_stream***, true)**
  - **PrintStream(***output_stream***, true, *charset_name*)**
  - **PrintStream(***file_name***)**
  - **PrintStream(***file_name***, *charset_name*)**

- **Generally, a PrintStream instance will not automatically flush the stream after: 1) a byte array is written, 2) a** println **method is invoked, nor 3) when a newline character is encountered**
  - **But if the constructor used includes 'true', the stream will automatically invoke the** flush() **method in those circumstances**

- **If no *charset_name* is specified, the underlying platform's default character set is used (EBCDIC for z/OS, of course)**
  - **Those constructors that mention a character set may throw an UnsupportedEncodingException**

# Using The PrintStream Class for File Output, 2

❒ **Another interesting point: a PrintStream will never throw an IOException!**

- ♦ **However, in some circumstances it will set an internal flag that the programmer can check, set, and reset using methods for the instance**

❒ **Since the result of writing PrintStream unique methods put the results out in the host platform's default character encoding (unless specifcially overridden when the PrintStream is instantiated), the output may not be readable on other platforms**

- ♦ **For example, under z/OS the output is in EBCDIC, by default**
- ♦ **If the data will only be used on a z/OS system, no problem**
  - ✗ But if it might be used on non-EBCDIC platforms, you had better either use a different charset or work with a Writer class instead of an OutputStream class
  - ✗ The docs have a caution: " All characters printed by a PrintStream are converted into bytes using the platform's default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes."
    - ➢ We examine the PrintWriter class later in this lecture

# Using The PrintStream Class for File Output, 3

❒ **The majority of the methods available to the PrintStream class, in addition to the methods it inherits, are:**

- **print(*boolean*)**
- **print(*char*)**
- **print(*char_array*)**
- **print(*double*)**
- **print(*float*)**
- **print(*int*)**
- **print(*long*)**
- **print(*object*)**
- **print(*string*)**
- **printf() - discussed later, for formatting**

- **println() - newline character appended to the end of the data**
- **println(*boolean*)**
- **println(*char*)**
- **println(*char_array*)**
- **println(*double*)**
- **println(*float*)**
- **println(*int*)**
- **println(*long*)**
- **println(**object**)**
- **println(*string*) - this is our familiar System.out.println(...)**

# Using The PrintStream Class for File Output, 4

❒ So the relevant code fragments might look like this:

```
try {
   PrintOut.print((testItem4.getPno()+blanks).substring(0,9));
   PrintOut.print((testItem4.getDesc()+blanks).substring(0,30));
   workQOH = zeros+Integer.toString(testItem4.getQOH());
   workOff = Integer.toString(testItem4.getQOH()).length();
   workQOH = workQOH.substring(workOff);
   PrintOut.print(workQOH);
   workUnPr = zeros+Float.toString(testItem4.getUnPr())+"0";
   workIndex = workUnPr.indexOf(".");
   workOff = workIndex - 7;
   workUnPr = workUnPr.substring(workOff,workIndex+3);
   PrintOut.print(workUnPr);
}
 catch (FileNotFoundException ex) {;}
```

❒ One could also instantiate the PrintStream this way:

```
filename = testItem4b.getPno();
PrintStream PrintOutb =
     new PrintStream(new FileOutputStream(filename));
```

- ♦ Or, of course, the constructor could specify a character set name, such as:

```
filename = testItem4a.getPno();
PrintStream PrintOuta =
   new PrintStream(filename, "utf16");
```

# Using The PrintStream Class for File Output, 5

❒ **And the results are ...**

- **When using the default character set:**

```
PART00405Hogwosh                                  0000000010000012.50
```

- **When using utf16 for the character set, for a different record:**

```
Ú..&. .ê.è..........<.?.Å.Ï.?.Ë.Ç.....................
```

✗ Or, after **dis utf16**

```
.PART00406Logwosh                                 0000000020000012.51
```

➢ Notice there is only one BOM here, at the begining of the file (as shown by the leading dot)

# Using The FileOutputStream Class for File Output

❒ **The FileOutputStream class extends OutputSteam and implements its methods, plus some of its own**

**Constructors**

- **FileOutputStream(***File_object***)**
- **FileOutputStream(***File_object***, {true|false})**
- **FileOutputStream(***FileDescriptor_object***)**
- **FileOutputStream(***file_name***)**
- **FileOutputStream(***file_name***, {true|false}))**
- **In the second and last constructors, if** true **is specified, then writes append to the end of the file; if** false **is specified, then writes begin at the front of the file**

**Example**

```
FileOutputStream FileRecOut =
                 new FileOutputStream(filename);
```

# Using The FileOutputStream Class for File Output, 2

- Recall that all the other constructors built off a FileOutputStream instantiation, but here it is just built off its own class

- Although there are a few methods unique to this class, we were happy to use the constructs we used earlier (see page 289), for example:

```
  workPartNo = testItem1.getPno() + blanks;
  workPartNo = workPartNo.substring(0,9);
  FileRecOut.write(workPartNo.getBytes("ISO-8859-1"));
.
.
.
```

  - and so on

- The results here mirror our earlier results using codepages (pages 290-291)

- The Java docs recommend using FileOutputStream (perhaps inside a BufferedOutputStream) for writing files that are truly just byte streams, such as image, audio, video, PDF documents, and so on

  - For writing fields, especially character fields, use the Writer classes

# Writer Classes for File Output

- **Now we examine Writer classes, as used for writing to HFS**
- **And we begin with a short summary of the methods available in the parent (abstract) class, since all the subclasses, of course, inherit these methods**

**Writer methods**

- **append(*char*) - append a single character to the stream**
- **close() - flush the stream then free the resources**
- **flush() - write remainder of any buffer**
- **write(*int*) - send the low order 8 bits of *int* to the stream as a byte**
- **write(*char_array*) - write the character array to the stream**
- **write(*char_array*, *strt*, *len*) - write part of the character array to the stream (begin at the *strt*-th entry, for *len* characters)**
- **write(*string*) - write the string to the stream**
- **write(*string*, *strt*, *len*) - write part of the string to the stream (begin at the *strt*-th entry, for *len* characters)**

# Using the BufferedWriter Class for File Output

❒ **The BufferedWriter class extends Writer and implements its methods**

**Constructors**

- **BufferedWriter(***writer***)**
- **BufferedWriter(***writer*, *buf_size***)**

**Example**

```
InventoryItem testItem2 =
  new InventoryItem("PART01205","Hogwesh",1,12.50f);

filename = testItem2.getPno();
BufferedWriter bfw2 =
 new BufferedWriter(new FileWriter(filename), 200);
```

- **Which creates a BufferedWriter instance in the same manner that we've been working with BufferedOutputStream classes**
  - ✗ In both examples we chose a buffer size of 200, just to show how it might look (our largest record is about 122 bytes)
  - ✗ If buf_size < 0, the constructor throws IllegalArgumentException

# Using the BufferedWriter Class for File Output, 2

- The methods available for this class are the same as for the parent class, plus this additional method:
  - newLine() - insert a newline character in the stream
- In preparation for our testing, we defined some variables at the top of our main method:

```
String filename;
String blanks = "                                        ";
String zeros = "000000000";

String pno;
String dsc;
String qoh;
String unp;

int pnoLen = 9;
int dscLen = 30;
int workOff;
int workIndex;
```

  - And, of course, we instantiated separate InventoryItem and BufferedWriter objects for each test (not shown, but the pattern should be clear by now)

# Using the BufferedWriter Class for File Output, 3

❒ **We tested the write(*string*, *strt*, *len*) and write(*string*) methods:**

```
pno = testItem2.getPno() + blanks;
dsc = testItem2.getDesc() + blanks;
qoh = zeros+Integer.toString(testItem2.getQOH());
workOff =
    Integer.toString(testItem2.getQOH()).length();
qoh = qoh.substring(workOff);
unp = zeros+Float.toString(testItem2.getUnPr())+"0";
workIndex = unp.indexOf(".");
workOff = workIndex - 7;
unp = unp.substring(workOff,workIndex+3);

bfw2.write(pno, 0, pnoLen);
bfw2.write(dsc, 0, dscLen);
bfw2.write(qoh, 0, qoh.length());
bfw2.write(unp, 0, unp.length());
bfw2.close();


pno = (testItem2a.getPno()+blanks).substring(0,pnoLen);
dsc = (testItem2a.getDesc()+blanks).substring(0,dscLen);
qoh = zeros + Integer.toString(testItem2a.getQOH());
workOff =
    Integer.toString(testItem2a.getQOH()).length();
qoh = qoh.substring(workOff);
unp = zeros+Float.toString(testItem2a.getUnPr())+"0";
workIndex = unp.indexOf(".");
workOff = workIndex - 7;
unp = unp.substring(workOff,workIndex+3);

bfw2a.write(pno);
bfw2a.write(dsc);
bfw2a.write(qoh);
bfw2a.write(unp);
bfw2a.close();
```

- ♦ **The second method is somewhat simpler and suits our needs**

# Using the BufferedWriter Class for File Output, 4

❒ **We also tested the write(*char_array*) method:**

```
pno = (testItem2b.getPno()+blanks).substring(0,pnoLen);
dsc = (testItem2b.getDesc() +
                              blanks).substring(0,dscLen);
qoh = zeros + Integer.toString(testItem2b.getQOH());
workOff =
      Integer.toString(testItem2b.getQOH()).length();
qoh = qoh.substring(workOff);
unp = zeros+Float.toString(testItem2b.getUnPr())+"0";
workIndex = unp.indexOf(".");
workOff = workIndex - 7;
unp = unp.substring(workOff,workIndex+3);

bfw2b.write(pno.toCharArray());
bfw2b.write(dsc.toCharArray());
bfw2b.write(qoh.toCharArray());
bfw2b.write(unp.toCharArray());
bfw2b.close();
```

- **Notice the use of the** toCharArray() **method, after converting the** int **and** float **items to** String

❒ **Of course, everything was wrapped in** try **and** catch **blocks**

# Using the BufferedWriter Class for File Output, 5

- The outputs we created, under obrowse:

```
PART01205Hogwesh                          0000000010000012.50
PART01206Logwesh                          0000000020000012.51
PART01207Sogwesh                          0000000030000012.52
```

  - So all character, all EBCDIC (since that is the native encoding of the underlying platform)

# Using the OutputStreamWriter Class for File Output

❒ **The OutputStreamWriter class is a bridge between character streams and byte streams; it extends Writer and implements its methods**

**Constructors**

- **OutputStreamWriter(***output_stream***)**
- **OutputStreamWriter(***output_stream***, *charset_object*)**
- **OutputStreamWriter(***output_stream***, *charsetEncoder_object*)**
- **OutputStreamWriter(***output_stream***, *charset_name*)**

**Notes**

- **Notice the underlying stream is an output stream not a Writer stream**
- **We will not discuss character set objects or character set encoders in this class, but we do deal with character sets by name (as already seen)**
- **It's possible to throw an UnsupportedEncodingException by those constructors that specify a character set**

# Using the OutputStreamWriter Class for File Output, 2

**Example**

```
InventoryItem testItem3 =
  new InventoryItem("PART01305","Hogw1sh",1,12.50f);

filename = testItem3.getPno();
OutputStreamWriter RecOut =
  new OutputStreamWriter(
                    new FileOutputStream(filename)
                         );
```

- Again, watch the line breaks

- Also, the Java docs suggest "wrapping around" a BufferedWriter to improve performance, and we find this works:

```
BufferedWriter RecOutc =
      new BufferedWriter(
               new OutputStreamWriter(
               new FileOutputStream(filename)
                                     )
                         );
```

- And this, indeed, works; an alternative approach to coding, similar to what we did on page 250, is this (which also works):

```
FileOutputStream    fos = null;
OutputStreamWriter  osw = null;
BufferedWriter      RecOutc;

fos      = new FileOutputStream(filename);
osw      = new OutputStreamWriter(fos);
RecOutc  = new BufferedWriter(osw);
```

# Using the OutputStreamWriter Class for File Output, 3

- The methods available for OutputStreamWriter instances are the same as for Writer, plus this:
  - getEncoding() - returns a String containing the character set name beng used

- So the code we used in our testing was similar to what we've shown already for BufferedWriter, and the results appear as:

```
PART01305Hogwish                          0000000010000012.50
PART01306Logwish                          0000000020000012.51
PART01307Sogwish                          0000000030000012.52
PART01308Togwish                          0000000040000012.53
```

  - As expected and desired

# Using The FileWriter Class for File Output

❒ **The FileWriter class extends OutputSteamWriter and implements its methods**

**Constructors**

- **FileWriter(***File_object***)**
- **FileWriter(***File_object***, {true|false})**
- **FileWriter(***FileDescriptor_object***)**
- **FileWriter(***file_name***)**
- **FileWriter(***file_name***, {true|false}))**
- **In the second and last constructors, if** true **is specified, then writes append to the end of the file; if** false **is specified, then writes begin at the front of the file**
- **All the constructors except the third one above can throw an IOException**

**Example**

```
FileWriter FileRecOut =
             new FileWriter(filename);
```

# Using The FileWriter Class for File Output, 2

- This class has the same methods and uses the same code as the earlier Writer classes
  - The main difference is the constructor for a FileWriter instance can just use its own class
- The results are consistent with what we've gotten with the other classes, too

# Using The PrintWriter Class for File Output

❒ **The PrintWriter class extends Writer and implements its methods, plus some of its own**

**Constructors**

- **PrintWriter(***File_object***)**
- **PrintWriter(***File_object***, *charset_name*)**
- **PrintWriter(***output_stream***)**
- **PrintWriter(***output_stream***, true)**
- **PrintWriter(***file_name***)**
- **PrintWriter(***file_name***, *charset_name*)**
- **PrintWriter(***writer***)**
- **PrintWriter(***writer***, true)**

❒ **Generally, a PrintWriter instance will not automatically flush the stream after: a** println**,** printf**, or a format method is invoked**

- **But if the constructor used includes 'true', the stream will automatically invoke the** flush() **method in those circumstances**

❒ **If no** *charset_name* **is specified, the underlying platform's default character set is used (EBCDIC for z/OS, of course)**

❒ **If an invalid character set is specified, an UnsupportedEncodingException is thrown**

❒ **Those constructors that include a file name can throw a FileNotFoundException**

# Using The PrintWriter Class for File Output, 2

## Sample constructors

- **Here are a couple of PrintWriter constructors we used in testing:**

```
PrintWriter PrintOut =
    new PrintWriter(new FileWriter(filename));

PrintWriter PrintOutd =
    new PrintWriter(new FileOutputStream(filename));
```

- **The first built directly off a file name (*writer*), the second built off a FileOutputStream (*output_stream*)**

❒ **As with the PrintStream we discussed earlier, a PrintWriter method will never throw an IOException!**

- **However, in some circumstances it will set an internal flag that the programmer can check, set, and reset using methods for the instance**

- **Note also that some of the constructors may possibly throw exceptions**

# Using The PrintWriter Class for File Output, 3

❒ **The majority of the methods available to the PrintWriter class, in addition to the methods it inherits, are:**

- **checkError() - return boolean status of error flag**
- **clearError() - sets error flag to false**
- **setError() - sets error flag to true**
- **format() - discussed later, under formatting**
- **print(*boolean*)**
- **print(*char*)**
- **print(*char_array*)**
- **print(*double*)**
- **print(*float*)**
- **print(*int*)**
- **print(*long*)**
- **print(*object*)**
- **print(*string*)**
- **printf() - discussed later, for formatting**
- **println() - inserts line separator string (usually newline)**
- **println(*boolean*)**
- **println(*char*)**
- **println(*char_array*)**
- **println(*double*)**
- **println(*float*)**
- **println(*int*)**
- **println(*long*)**
- **println(**object**)**
- **println(*string*)**

# Using The PrintWriter Class for File Output, 4

❒ Here are a couple of the tests we ran:

```
filename = testItem4.getPno();
   // create print writer off FileWriter
   // methods print(string), print(int), print(float)
PrintWriter PrintOut =
       new PrintWriter(new FileWriter(filename));
PrintOut.print((testItem4.getPno()+blanks).
                                        substring(0,pnoLen));
PrintOut.print((testItem4.getDesc()+blanks).
                                        substring(0,dscLen));
qoh = zeros+Integer.toString(testItem4.getQOH());
workOff = Integer.toString(testItem4.getQOH()).length();
qoh = qoh.substring(workOff);
PrintOut.print(qoh);
unp = zeros+Float.toString(testItem4.getUnPr())+"0";
workIndex = unp.indexOf(".");
workOff = workIndex - 7;
unp = unp.substring(workOff,workIndex+3);
PrintOut.print(unp);
PrintOut.close();


filename = testItem4a.getPno();
   // create print writer off filename
   // method print(string) containing all concatenated
PrintWriter PrintOuta = new PrintWriter(filename);
pno = (testItem4a.getPno()+blanks).substring(0,pnoLen);
dsc = (testItem4a.getDesc()+blanks).substring(0,dscLen);
qoh = zeros + Integer.toString(testItem4a.getQOH());
workOff = Integer.toString(testItem4a.getQOH()).length();
qoh = qoh.substring(workOff);
unp = zeros + Float.toString(testItem4a.getUnPr())+"0";
workIndex = unp.indexOf(".");
workOff = workIndex - 7;
unp = unp.substring(workOff,workIndex+3);
PrintOuta.print(pno + dsc + qoh + unp);
PrintOuta.close();
```

# Using The PrintWriter Class for File Output, 5

- **The results are all in EBCDIC, in the same vein as we would expect (all character strings)**

- **One slightly surprising outcome is this example**

```
PrintOuta.print(testItem4f.getPno()+
                    testItem4f.getDesc()+
                    testItem4f.getQOH()+
                    testItem4f.getUnPr());
```

  - **Surprising because getQOH() returns int, and getUnPr() returns float, yet the encompassing print method automatically converts them to String / characters**

    - ✗ Unfortunately, not fixed length strings, which is essential in reading the data back

# Summary and Comparison

- **Here, in a nutshell, is what we found, hitting some of the differences and similarities**

  - **FilterOutputStream uses write(*int*), write(*byte*), write(*byte_array*), write(*byte_array*,*start*,*len*) methods; when writing with charset of utf16, there is a BOM at the front of each field; constructor must be off an output stream**

  - **BufferedOutputStream works same as FilterOutputStream, presumably with better peformance**

  - **DataOutputStream uses writeChars(*string*), writeInt(*int*), writeFloat(*float*), and so on; results in utf16 with no BOM for character and string, internal format for numeric data types; constructor off a FileOutputStream or a BufferedOutputStream**

  - **PrintStream uses print(*string*), print(*int*), print(*float*), println(*string*), println(*int*), println(*float*), and so on; if specify utf16 in the PrintStream constructor, there is a single BOM for the whole file; otherwise, default encoding (EBCDIC) used; constructor may be off a File object, a file name, or an output stream; never throws an IOException**

  - **FileOutputStream uses write() methods same as FilterOutputStream; when writing with utf16, a BOM at the front of each field; constructor is off a file name, a file descriptor, or a File object; can specify append in constructor**

# Summary and Comparison, 2

- **Here, in a nutshell, is what we found, hitting some of the differences and similarities, continued**
  - **BufferedWriter uses write(*string*[,*start*,*len*]), write(*char_array*), write(*int*), write(*char_array*[,*start*,*len*]) , append (*char*) methods; output in local encoding; constructor off a writer**
  - **OutputStreamWriter uses methods same as BufferedWriter; constructor off an output stream, may specify character set in constructor**
  - **FileWriter uses same methods as BufferedWriter; constructor off file name, file descriptor, or File object; can specify append in constructor**
  - **PrintWriter uses print(*string*), print(*int*), print(*float*), println(*string*), println(*int*), println(*float*), and so on, methods; constructor off writer, output stream, file name, or File object; can specify character set in constructor; when constructor uses utf16, results in file with one BOM**

# Summary and Comparison, 3

- When you need to create files in the HFS, consider what type of output you want and what information you will have on hand at the time you create, write to, or append to, the file, to narrow down your choices
  - One general rule of thumb: when writing just bytes, use output stream classes, when writing out character data, use writer classes
- For the case study labs, we decided to write string data out in UTF16 (so we can support any Unicode character and work with it) and numeric data types (in particular, int and float) in internal format
  - Which leads us to use the DataOutputStream class as our preferred approach for the case study
- There may well be times when you chose one or more of the other options to meet the needs of your applications

## Computer exercise: Writing HFS Files from Java

In your J510 directory are the two files we used for testing and experimenting:

testIO.java - tests OutputStream classes

testWriters.java - tests Writer classes

These files provide tools you can use to test and experiment with, and also code you can cut an paste into our InventoryDataManager class, as appropriate.

1. Compile and run testIO.java; this will produce the following files:

| Filename | Characteristics | |
|---|---|---|
| PART00105 | ISO-8859-1 character | from FilterOutputStream |
| PART00106 | CP037 character | " |
| PART00107 | UTF16 character | " |
| PART00205 | ISO-8859-1 character | from BufferedOutputStream |
| PART00206 | CP037 character | " |
| PART00207 | UTF16 character | " |
| PART00305 | UTF16 character, int, and special float; DataOutputStream | |
| PART00306 | UTF16 character, int, and special float; DataOutputStream | |
| PART00307 | UTF16 character, int, and special float; DataOutputStream | |
| PART00405 | CP037 character | from PrintStream |
| PART00406 | UTF16 character | " |
| PART00407 | CP047 character | " |
| PART00505 | ISO-8859-1 character | from FileOutputStream |
| PART00506 | CP037 character | " |
| PART00507 | UTF16 character | " |

2. Compile and run testWriters.java; this will produce the following files:

| Filename | Characteristics | |
|---|---|---|
| PART01205 | CP037 character | from BufferedWriter |
| PART01206 | CP037 character | " |
| PART01207 | CP037 character | " |
| PART01208 | CP037 character | " |
| | (w/ extra character appended in PartNo field) | |
| PART01305 | CP037 character | from OutputStreamWriter |
| PART01306 | CP037 character | " |
| PART01307 | CP037 character | " |
| PART01308 | CP037 character | " |
| PART01405 | CP037 character | from PrintWriter |
| PART01406 | CP037 character | " |
| PART01407 | CP037 character | " |
| PART01408 | CP037 character | " |
| PART01409 | CP037 character | " |
| PART01410 | UTF16 character | " |
| PART01505 | CP037 character | from FileWriter |
| PART01506 | CP037 character | " |

*** more on next page ***

Computer Exercise, p. 3.

Now we have some data to work with in later labs. And now we will also have you make some new data using InventoryDataManager. Instead of displaying the data, you should write it to a file for later use.

We have made the application design decision to use DataOutputStreams for writing to DASD, putting String data in Unicode and numeric data in native format. So ...

3. Modify the InventoryDataManager class:

   * add a new method, writeUTF16(), that creates an HFS file with a name the same as workPartno and that contains the stream: workPartno, workDesc, workQOH, and workUnPr based on the lecture points on page 294-297.

     remember to use **try** and **catch** blocks

   * in the logic for handling the menu choices, for case 1, replace the four System.out.println() commands with the single invocation of writeUTF16()

4. Compile and run InventoryDataManager:

   * create two new Inventory Items:

| | | |
|---|---|---|
| Part number | PartyOver | PART04000 |
| Description | Have a good time | Yellow shirts (large) |
| Quantity on hand | 12 | 22 |
| Unit price | 2212.12 | 45.33 |

5. Browse the resulting files to ensure the data is what you expect.

# Section Preview

- **Case Study Part 5**
  - **Review of our classes so far**
  - **DataManager Re-Examined**
  - **Early exit from process**
  - **The String class**
  - **Early exit from process (Machine Exercise)**

# Review of Classes so Far

❒ **The classes we create as we build our application are our tools, and as you see, we tend to create lots of classes and methods - we need to keep track of them**

- **Here's a quick review of the classes and methods we have in place for our application so far (not counting classes we've just used for experiments)**

**InventoryItem**

- **getPno() / setPno(*part_no*)**
- **getDesc() / setDesc(*description*)**
- **getQOH() / setQOH(*quantity*)**
- **getUnPr() / setUnPr(*price*)**
- **calcValue()**

**Foodstuffs - subclass of InventoryItem**

- **Inherited methods from InventoryItem**
- **getExpDate() / setExpDate(*date*)**

**InvRept**

- **main()**

**htmlGen**

- **putStart(), putEnd()**
- **putCol(*column_value*)**

# Review of Classes so Far, 2

❒ **Classes so far, continued**

**InventoryDataManager**

- **main()**
- **getFields()**
- **writeUTF16()**

**DataManager**

- **PromptAndGetString(***prompt*, *BufferedReader***)**
- **PromptAndGetInt(***prompt*, *BufferedReader***)**
- **PromptAndGetFloat(***prompt*, *BufferedReader***)**

❒ **We now have the ability to prompt for data and save the resulting object in a file in the HFS**

- **As well as generating some HTML**

❒ **It might seem natural to follow this with some methods for retreiving an object's values from its HFS file**

- **But it turns out we have more urgent issues first**

  ✗ One of which is how to make this process more user friendly

# DataManager Re-Examined

- **Perhaps you have already encountered this: during data entry you have made a mistake and need to go back and correct it**
  - **Or you need to stop what you're doing and come back later**
  - **We need to provide the user some way to do this**

# Early Exit From Process

- **Using the philosophy that decisions should be made as close to the source of information as possible, it seems reasonable that the place we ask the user for data is the place we should give him or her the option to discontinue the operation**
  - **Data is coming in as a character string, so why not allow some string to indicate an action such as 'quit' instead of data?**
  - **If the user enters a string one character long, first examine it to see if it might be, say:**
    - c - meaning cancel the process, meaning the current data entry step should be cancelled
      - That is, none of the data should be saved and control should return to the main logic
        - Note that the PromptAndGet routine can't do anything about this, except pass the request back to the invoker, since that is where the looping action is
  - **So we establish this convention:**
    - At data entry time, if the user enters a one-character string containing "c", for any field:
      - For String methods we return the null string
      - For numeric methods we return a value of -1 in the numeric format requested (int, float, double, long, etc.)

# Early Exit From Process, 2

- **This will require changes in two classes:**
  - **The DataManager class methods will have to detect the entry of a one-byte String containing "c" and respond appropriately**
  - **The InventoryDataManager class getFields() method will have to check the returned value to determine if it should continue on or not**

# Early Exit From Process, 3

❒ **In DataManager, we find we need these changes:**

**The prompt in each method needs to change from**

```
System.out.println("Please enter the value you want for "
                   + text + ": ===>"):
```

**To**

```
System.out.println("Please enter the value you want for "
                   + text + ", or 'c' to cancel: ===>"):
```

**Each routine must examine response for a single character "c"**

✗ The intuitive code might be something like this:

```
if (stringIn == "c") {stringIn = null;}
```

✗ But it turns out this doesn't work as we would like

✗ The reason is: String data are not primitives but objects and we cannot compare objects simply using the == operator

➢ Two objects being equal means they are the same object

➢ We need to use various "compareTo" or "equals" methods to make the tests we are looking for

➢ We've seen this before, but now's the time to get more precise

❒ **Yes, there's a digression coming on ...**

# The String Class

❒ **We mentioned numerous times that String is a special class**

- ♦ **And we've even had places where we have referenced / used various String methods (for example, pages 95, 105-110, 215, 255, 281-282, and 288)**

❒ **This seems like a good place to cover the capabilities of this important class in more detail**

**Fields**

- ♦ **The String class has only one field, called CASE_INSENSITIVE_ORDER**
- ♦ **This is defined as a "Comparator that orders String objects as by compareToIgnoreCase"**
  - ✗ There doesn't seem to be a definition of what "Comparator" field is anywhere in the specs; so we'll just ignore this and go on
    - ➢ Note that there is a Comparator interface but, of course, we have not discussed interfaces yet

# The String Class, 2

## Constructors

❒ **There are 13 non-deprecated constructors:**

- **String() - create an empty String object**
- **String(*string*) - create a copy of string; marginal value**
- **String(*char_array[]*) - create a String that represents the sequence of characters contained in the argument**
- **String(*char_array[]*, *off*, *count*) - creates a String representing a subset of a character array**
- **String(*int_array[]*, *off*, *count*) - creates a String containing a subset of an array of Unicode codepoints stored as integers**
- **String(*byte_array[]*, *off*, *count*, *charsetName*) - creates a String that is a subset of a byte array, using the named character set to interpret the input**
- **String(*byte_array[]*, *off*, *count*, *charset*) - creates a String that is a subset of a byte array, using the provided character set to interpret the input**
- **String(*byte_array[]*, *charsetName*) - creates a String that contains the contents of a byte array, using the named character set to interpret the input**

# The String Class, 3

## Constructors, continued

❒ **There are 13 non-deprecated constructors, continued:**

- ♦ **String(***byte_array[]***, *charset*) - creates a String that contains the contents of a byte array, using the provided character set to interpret the input**
- ♦ **String(***byte_array[]***, *off*, *count*) - creates a String from a subset of a byte array, using the default character set to interpret the input**
- ♦ **String(***byte_array[]*) - creates a String from a byte array, using the default character set to interpret the input**

## Methods

❒ **There are 65 methods available (most useful highlighted):**

- ♦ *string*.charAt(*index*) - returns the character at location *index* (an integer value) in *string***
- ♦ *string*.codePointAt(*index*) - returns the Unicode code point at location *index* (an integer value) in *string*, as an** int
- ♦ *string*.codePointBefore(*index*) - returns the Unicode code point before location *index* (an integer value) in *string*, as an** int
- ♦ *string*.codePointCount(*strt*, *end*) - returns the number of Unicode code points between *strt* and *end* (integer values) in *string*, as an** int

# The String Class, 4

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**hashCode() - returns an integer hash code for** *string*
- *string*.**equals(***other_string***) - returns a boolean** true **or** false**;** true **if the argument is a non-null String object that represents the same sequence of characters as this object**
- *string*.**equalsIgnoreCase(***other_string***) - returns a boolean** true **or** false**;** true **if the argument is a non-null String object that represents the same sequence of characters as this object; compare is case insensitive**
- *string*.**compareTo(***other_string***) - returns an integer: zero if the strings are equal; a negative number if** *string* **precedes** *other_string***; a positive number if** *string* **follows** *other_string***;**
- *string*.**compareToIgnoreCase(***other_string***) - returns an integer: zero if the strings are equal; a negative number if** *string* **precedes** *other_string***; a positive number if** *string* **follows** *other_string***; the comparison is case insensitive**

  ✗ Note string compares are lexicographical: at the first position with different characters, the string with its corresponding codepoint less than the other is said to precede the other string; if the strings are of different lengths and have equal values up to their common length, the shorter string is determined to precede the longer string

# The String Class, 5

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**length()** - **returns the length (number of Unicode code units) in the** *string*, **as an integer**
- *string*.**isEmpty() - returns true if length of** *string* **is 0**
- *string*.**substring(***strt***)** - *string*.**substring(***strt*, *end***) - returns a String containing the characters beginning at location** *strt* **through location** *end*; **if** *end* **is not specified, implies "to end of string"**
- *string*.**toLowerCase()** - *string*.**toLowerCase(***locale***) - converts all characters of** *string* **to lower case, based on the default locale or a specified** *locale*
- *string*.**toUpperCase()** - *string*.**toUpperCase(***locale***) - converts all characters of** *string* **to upper case, based on the default locale or a specified** *locale*
- *string*.**trim() - returns a copy of** *string* **with leading and trailing whitespace removed**
- *string*.**toString() - returns itself**
- *string*.**concat(***string_2***) - returns a String that is the concatenation of the instance** *string* **followed by the argument** *string_2*; **result is the same as using the "+" operator**

# The String Class, 6

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**indexOf(***char***) -**
  *string*.**indexOf(***string_2***) - returns location of first occurrence of** *char* **or** *string_2* **in** *string*

- *string*.**indexOf(***char***, *strt*) -**
  *string*.**indexOf(***string_2***, *strt*) - returns location of first occurrence of** *char* **or** *string_2* **at or after location** *strt* **in string; the returned value is from the start of** *string*, **not from the** *strt* **offset**

- *string*.**lastIndexOf(***char***) -**
  *string*.**lastIndexOf(***string_2***) - returns location of last occurrence of** *char* **or** *string_2* **in** *string*

- *string*.**lastIndexOf(***char***, *strt*) -**
  *string*.**lastIndexOf(***string_2***, *strt*) - returns location of last occurrence of** *char* **or** *string_2* **at or before location** *strt* **in string**

  ✗ Note: all "locations" are calculated with "zero" being the first; if there is no location that satisfies the test, -1 is returned

- *string*.**startsWith(***string_2***) -**
  *string*.**startsWith(***string_2***, *strt*) returns** true **(boolean) if the initial character sequence of** *string* **[after location** *strt***] is the same as** *string_2*

- *string*.**endsWith(***string_2***) - returns** true **if the ending character sequence of** *string* **is the same as** *string_2*

# The String Class, 7

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**getBytes() - encodes** *string* **into a byte array using the current default charset**

- *string*.**getBytes(***charset***) - encodes** *string* **into a byte array using the** *charset* **object**

- *string*.**getBytes(***charset_name***) - encodes** *string* **into a byte array using the named charset**

  ✗ There is another **getBytes** method that is now deprecated

- *string*.**getChars(***strt***, *end*, *dest[]*, *begin*) - copies a subset** *string*, **from** *strt* **to** *end*, **into character array** *dest[]*, **beginning at index** *begin* **of dest[]**

- **String.copyValueOf(***char[]***) - returns a String that represents the character sequence in the input array**

- **String.copyValueOf(***char[]*, *strt*, *count***) - returns a String that represents the character sequence in the identified subset of the input array**

# The String Class, 8

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- ♦ *string*.**contentEquals(***string_buffer***) - returns** true **(boolean) if the value of** *string* **matches the value in** *string_buffer*

  - ✗ Technically, a StringBuffer is a thread-safe class designed to hold String data

  - ✗ String data is unchangeable: once a String is declared and initialized, it's size never changes(!)

    - ➢ When, for example, you concatenate one string to another, a whole new object is constructed

    - ➢ StringBuffers, on the other hand, are designed to be modified, so they have better performance if you are doing a lot of assignments or modifications to String variables, consider using StringBuffer variables instead

# The String Class, 9

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**contentEquals(***char_sequence***) - returns** true **(boolean) if the value of** *string* **matches the value in** *char_sequence*

- *string*.**contains(***char_sequence***) - returns** true **(boolean) if** *string* **contains the value in** *char_sequence* **somewhere in the String**

  ✗ Technically, a CharSequence is an interface representing "a readable sequence of char values"

## Notes

- **You can declare variables of type** CharSequence **exactly as you declare a variable of type String, while a variable of type** Stringbuffer **has to be declared and instantiated; for example:**

```
CharSequence newstack = "qwertyuiop";
StringBuffer newstock;
             newstock = new StringBuffer("asdfghjk");
```

- **Experiment verifies that you can use a String variable in** contains **and** contentEquals **methods**

# The String Class, 10

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- *string*.**subSequence(***strt*, *end*) **- returns a CharSequence containing a subset of** *string*

- *string*.**offsetByCodePoints(***strt*, *offset*) **- returns, as an integer, the offset within** *string* **from position** *strt*; **the returned value is set as a number of codepoints (where surrogate pairs count as one codepoint, but each unpaired surrogate counts as one codepoint) [too esoteric to even think about]**

- *string*.**regionMatches({**true|false**},** *strt*, *string_2*, *start*, *len*) **- returns true if** *string*.**substring(***strt*, *len*) **=** *string_2*.**substring(***start*, *len*)**, where the compare is case-sensitive if the first argument is** false **and case-insensitive if the first arguent is** true

- *string*.**regionMatches(***strt*, *string_2*, *start*, *len*) **- returns true if** *string*.**substring(***strt*, *len*) **=** *string_2*.**substring(***start*, *len*)**; the compare is always case-sensitive**

- *string*.**replace(***old_char*, *new_char*) **- replace in** *string* **all occurrences of** *old_char* **with** *new_char*

- *string*.**replace(***old_charseq*, *new_charseq*) **- replace in** *string* **all occurrences of CharSequence** *old_char* **with CharSequence** *new_char*

- *string*.**intern() - returns a String containing the canonical representation for** *string* **(a Unicode issue, mostly for non-Western codepoints)**

# The String Class, 11

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- ♦ **String.valueOf(*boolean*) - returns the String representation of** *boolean*
- ♦ **String.valueOf(*char*) - returns the String representation of the single character** *char*
- ♦ **String.valueOf(*char[]*) - returns the String representation of the array of characters** *char[]*
- ♦ **String.valueOf(*char[], strt, end*) - returns the String representation of the character array** *char[]*, **from** *strt* **to** *end*
- ♦ **String.valueOf(*double*) - returns the String representation of** *double*
- ♦ **String.valueOf(*float*) - returns the String representation of** *float*
- ♦ **String.valueOf(*int*) - returns the String representation of** *int*
- ♦ **String.valueOf(*long*) - returns the String representation of** *long*
- ♦ **String.valueOf(*object*) - returns the String representation of** *object*

✗ Most of the batch on this page are implicitly invoked when you include a variable of that type on a System.out.println() statement

# The String Class, 12

## Methods, continued

❒ **There are 65 methods available (most useful highlighted), continued:**

- ♦ **At this point, there are these groups of methods left unexamined:**
  - ✗ Two methods called **format** for using C-like format strings to direct how to display data
    - ➢ Postponed until a later section
  - ✗ Methods **replaceAll**, r**eplaceFirst**, and two variations on **split**
  - ✗ These all use r**egular expressions**, which we'll discuss in the next section on data validation

❒ **We are now ready to get back to our main thrust for this section: how to take an early exit from a process ...**

# Early Exit From Process, discussion resumed

❒ **Recall, we said that in DataManager, we find we need these changes:**

**The prompt in each method needs to change from**

```
System.out.println("Please enter the value you want for "
                   + text + ": ===>"):
```

**To**

```
System.out.println("Please enter the value you want for "
                   + text + ", or 'c' to cancel: ===>"):
```

**Each routine must examine response for a single character "c"**

- **For PromptAndGetString:**

```
    if (stringIn.equals("c")) { stringIn = null; }
```

- **For PromptAndGetInt:**

```
    if (stringIn.equals("c")) { backInt = -1; }
    else { backInt = Integer.parseInt(stringIn); }
```

- **For PromptAndGetFloat:**

```
    if (stringIn.equals("c")) { backFloat = -1f; }
    else { backFloat = Float.parseFloat(stringIn); }
```

# Early Exit From Process, discussion resumed

❒ **The last piece here is in InventoryDataManager:**

**Need to define a static variable as a flag, maybe:**

```
static boolean writeOK = true;
```

✗ The intent is to only write a file if writeOK is true

**Next, in case 1 of the menu, we end up with this**

```
    getFields();
    if (writeOK) { writeUTF16(); }
    ; break;}
```

# Early Exit From Process, discussion concluded

**Finally, change getFields() from:**

```
workPartno = DataManager.PromptAndGetString(
                          "part number", rdr2);
workDesc   = DataManager.PromptAndGetString(
                          "description", rdr2);
workQOH    = DataManager.PromptAndGetInt(
             "initial quantity on hand", rdr2);
workUnPr   = DataManager.PromptAndGetFloat(
                           "Unit Price", rdr2);
```

**To**

```
writeOK   = true;

workPartno = DataManager.PromptAndGetString(
                          "part number", rdr2);
  if (workPartno == null) {writeOK = false;}

if (writeOK) {
  workDesc    = DataManager.PromptAndGetString(
                             "description", rdr2);
  if (workDesc == null) {writeOK = false;}
            }

if (writeOK) {
  workQOH     = DataManager.PromptAndGetInt(
               "initial quantity on hand", rdr2);
  if (workQOH < 0) {writeOK = false;}
            }

if (writeOK) {
  workUnPr    = DataManager.PromptAndGetFloat(
                   "Unit Price", rdr2);
  if (workUnPr < 0) {writeOK = false;}
            }
```

Computer exercise: Early Exit from Process

Set up your data entry application (InventoryDataManager) to allow the user to exit the process early by entering a single "c", for cancel, based on the lecture content:

1. Modify DataManager.java to change the prompt and to examine the response in order to handle an early exit; see page 346

2. Compile DataManager.java.

3. Modify InventoryDataManager.java to handle the possibility of an early exit request, based on pages 347-348

4. Compile InventoryDataManager.java

5. Test InventoryDataManager; in particular, be sure to test:
    * adding a full entry still works
    * starting a new entry but cancelling at various points works: that is, test the cancel option for each of unit price, quantity on hand, description, and part number.

**Optional exercise stretches:**

in DataManager, allow the user to enter an uppercase C or a lowercase c to cancel the process

in InventoryDataManager, add messages to the user to acknowledge the early termination

This page intentionally left almost blank.

# Section Preview

- **The Case Study Part 6**
  - **Data Validation**
  - **Validation logic**
  - **Regular expressions**
  - **Data Validation in DataManager**
  - **Object data validation (Machine Exercise)**

# Data Validation

❒ **Another issue that needs to be addressed early is how to ensure our data are valid**

❒ **At this point we consider the question: what are the criteria for data validity for our application?**

- ♦ **Consider each field:**

  - ✗ Part number: a string of the form PARTnnnnn where nnnnn represents five character numeric digits (that is, each n is in the range 0-9) for Inventory Items and FOODnnnnn for Foodstuffs

  - ✗ Description: any 30 valid Unicode characters, left justified, first character non-whitespace, padded on the right with Unicode spaces

  - ✗ Quantity on hand: an integer in the range of 0 to 2,147,483,648 (but in reality, never more than 1,000,000)

  - ✗ Unit price: a numeric value with two positions to the right of the decimal point and a value in the range of 0.01 to 1,000,000.00 (currency issues to be discussed later)

  - ✗ Expiration data (for Foodstuffs): initially an 8-character string in the form yyyymmdd, with the date representing at least one day later than the date the product is entered into the data base (alternative formats discussed later)

# Validation Logic

❒ **You can probably imagine at least a first cut of how to make these tests, and we explore them in detail soon, but first we ask these questions:**

- ♦ **Where do the tests belong?**
  - ✗ Since they are intimately connected with inventory itemrs, it would seem the tests could be comfortably placed as close to the point of data entry as possible, in the InventoryDataManager class, as part of the getFields() method
    - ➢ This way, we don't clutter up DataManager with tests that might not apply for all users of those methods
    - ➢ Still, we will enhance DataManager in other ways, regarding the numeric inputs; stay tuned
- ♦ **What logic should we follow if incoming data is does not pass these tests?**
  - ✗ Need to set some boolean variables: one for each field, initialized to **false i**n each case
  - ✗ As each field is entered by the user, the routine should validate that field: loop in the data entry logic until the user enters a valid value or cancels the process
  - ✗ At the end, if all the fields are valid, set writeOK to **true**, otherwise, set it to **false**

# Validation Logic, 2

❒ At the top of getFields(), then, we add:

```
boolean validPno   = false;
boolean validQOH   = false;
boolean validUnPr = false;
```

- ♦ Notice there is no check for the description field: any string is valid, and the writeUTF16 routine will ensure only the first 30 characters get written out

❒ Here's some psuedo-code for the logic for our revised getFields() method:

```
workPartno = DataManager.PromptAndGetString("part number", rdr2);
while (!validPno & writeOK)   {
      if (workPartno = null) { writeOK = false; }
      else { force workPartno to upper case
             if (workPartno is valid) { validPno = true; }
             else { display error message;
            workPartno = DataManager.PromptAndGetString("part number",
                                                          rdr2);}
           }    }

if (writeOK) { prompt and get for description;
               if (description = null) { writeOK = false; } }

while (!validQOH & writeOK) {
      workQOH = DataManager.PromptAndGetInt("quantity on hand", rdr2);
      if (workQOH < 0) { writeOK = false; }
      else { if (workQOH is valid) { validQOH = true; }
             else { display error message; }
           }    }

while (!validUnPr & writeOK) {
      workUnPr = DataManager.PromptAndGetFloat("unit price", rdr2);
      if (workUnPr < 0) { writeOK = false; }
      else { if (workUnPr is valid) { validUnPr = true; }
             else { display error message; }
           }    }
```

# Validation Logic, 3

- The validity tests for quantity on hand and unit price can be coded something like this:

**Quantity on hand**

```
if ((0 <= workQOH) & (workQOH <= 1000000))
        { validQOH = true; }
   else {
     System.out.println("Quantity not valid.");
     System.out.println("Must be an integer " +
           "between 0 and 1000000, inclusive.");
     System.out.println("Please try again...");
         }
```

**Unit price**

```
if ((0.01f <= workUnPr) & (workUnPr <= 1000000.00f))
         { validUnPr = true; }
   else {
     System.out.println("Price is not valid.");
     System.out.println("Price must be number " +
       "between 0.01 and 1000000.00, inclusive.");
     System.out.println("Please try again...");
         }
```

- But the validity test for part number requires either some very complicated coding or the use of regular expressions ...

# Regular Expressions

❒ **The central ideas behind regular expressions are those of patterns**

- ♦ **Examine a string to see if it looks like a picture, matches a pattern**
- ♦ **Make changes to characters in a string based on patterns**

❒ **In Java, the place to start is the** java.util.regex **package which contains two classes**

**Pattern**

- ♦ **Which includes eight fields (each a flag), no constructors, and ten methods**
  - ✗ Two of these methods are called **compile**, and these are how instances are created, not through classic constructors
  - ✗ Another method is called **matcher**, and it creates a Matcher instance

**Matcher**

- ♦ **Which includes no fields, no constructors (but, see** matcher **above), and 31 methods**
  - ✗ The method we care about here is **matches()** - it returns **true** if the string under examination matches the pattern in the matcher instance we created

# Regular Expressions, 2

❒ **The Pattern and Matcher classes are inextricable and mysteriously intertwined**

- ♦ **Between the two of them, you can:**
  - ✗ Find if a string matches a pattern
  - ✗ Split a String apart into a String array, with each element chosen based on a pattern found in the original String
  - ✗ Find the first, last or all occurrences of a pattern in a string (these occurrences are called regions)
  - ✗ And more, way beyond our scope here

# Regular Expressions, 3

- **In the big picture for our work, we want to see if the contents of a field (PartNo) matches the patterns we mentioned before: PART*ddddd* or FOOD*ddddd* where the *ddddd* represents five character numeric digits; the process to do this is:**
  - **Construct a regular expression that describes what we are looking for**
  - **Compile this expression into a Pattern instance (just called a** pattern**)**
  - **Construct a** matcher **(a Matcher instance) using the matcher() method on the Pattern instance and the string you are working with**
  - **Invoke the matches() method on the** matcher **to see if the result is** true **or** false

**Code sample**

```
Pattern pno =
        Pattern.compile("PART[0-9][0-9][0-9][0-9][0-9]");
Matcher m1 = pno.matcher(workPartNo);
if (m1.matches) {...}
```

- **Our work now is to describe how to construct regular expressions in Java**
  - **Then we briefly examine some alternatives for checking matches using patterns**

# Regular Expressions, 4

- regular expessions in Java are similar to the same construct in other langauges such as Perl, awk, python. and so on - but there are differences

- A regular expression is a string composed of ordinary and metacharacters

  - Ordinary characters simply represent themselves, as literals

  - Metacharacters represent special types, classes, or groups, of characters

# Regular Expressions, 5

❒ **Here are some basics:**

- **In Java code, regular expressions are contained in quoted strings**
- **Each ordinary character takes one one character position**
- **If a position is allowed to have any character from a list of characters, the list is put in brackets, for example:** [13579]
- **If the list of possibilties has no gaps, you may specify the starting and ending values separated by a dash,** *e.g.*: [0-9]

  ✗ Which explains our example earlier of **"PART[0-9][0-9][0-9][0-9][0-9]"**

- **A common way to represent "any uppercase or lowercase alphabetic letter" is:** [a-zA-Z]
- **Use a caret (^) to indicate "any character except these", for example:**

  **[^13579]** **any character except 1 or 3 or 5 or 7 or 9**

# Regular Expressions, 6

## ❒ More regular expression fundamentals

- **Characters may be represented as escape sequences ( \ - followed by character(s)):**
  - ✗ \0***n*** - an octal digit (0 <= ***n*** <=7)
  - ✗ \0***nn*** - a pair of octal digits (0 <= ***n*** <=7)
  - ✗ \0***mnn*** - a triplet of octal digits (0 <= ***m*** <= 3, 0 <= ***n*** <=7)
  - ✗ \x***hh*** - the character represented by those hex digits
  - ✗ \u***hhhh*** - the Unicode character represented by those hex digits
  - ✗ \t - the tab character (\u0009)
  - ✗ \n - the newline (line feed) character (\u000A)
  - ✗ \r - the carriage return character (\u000D)
  - ✗ \f - the form feed character (\u000C)
  - ✗ \a - the alert (bell sound) character (\u0007)
  - ✗ \\ - the backslash itself

- **Some classes of characters are used so often, there are abbreviations pre-defined for them, in escape sequences:**
  - ✗ . - any characters (dot; not an escaped character); use \. for an actual dot
  - ✗ \d - a digit (same as [0-9] )
  - ✗ \D - any character except a digit (same as [^0-9] )
  - ✗ \s - any whitespace character (same as [ \t\n\x0B\f\r] )
  - ✗ \S - any non-whitespace character (same as [^\s] )

# Regular Expressions, 7

❒ **Our earlier pattern for part number, then, could be:**

✗ "PART\d\d\d\d\d" - except for one little problem:

➢ inside the quoted string, the Java compiler will look at the escaped characters before the **compile()** method ever sees them - and Java finds these to be in error

♦ **The work around is you need to add an extra level of escaping:**

✗ "PART\\d\\d\\d\\d\\d" - does the trick

❒ **But what about the possibility of a number following the pattern FOOD*ddddd*?**

♦ **To solve this issue, we need to introduce capturing groups: regular expression fragments bounded by parentheses ( ( ) )**

♦ **We could, for example, code:**
**"(PART)\\d\\d\\d\\d\\d"**

♦ **The power from groups comes from the operators that can be applied - in our case, for example, the vertical bar represents "or", so we could code:**

✗ "((PART)|(FOOD))\\d\\d\\d\\d\\d"

➢ Notice the extra set of parentheses around the PART and FOOD group

# Regular Expressions, 8

❒ **There are some useful operators that can work on units or groups:**

- ♦ **Multipliers can be applied using braces ( { } ) as in:**
  - ✗ (***group***){***n***} - repeat ***group n*** times
  - ✗ (***group***){***n***,} - repeat ***group*** at least ***n*** times
  - ✗ (***group***){***n***,***m***} - repeat ***group*** at least ***n*** times, but no more than ***m*** times
- ♦ **So our regular expression could also be written as:**
  - ✗ "((PART)|(FOOD))\\d{5}"

**or also**

- ✗ "((PART)|(FOOD))[0-9]{5}""

# Regular Expressions, 8

- We mentioned the fields in the Pattern class, and we also mentioned they are all int fields used as flags
- The interpretation of the flags is as follows:

| Name (integer value) | Meanining |
|---|---|
| UNIX_LINES (1) | Restrict the characters recognized as line terminators to just \n |
| CASE_INSENSITIVE (2) | Makes compares case insensitive (only supported for US-ASCII characters) |
| COMMENTS (4) | Embedded whitespace and comments in patterns are ignored (comments begin with # ) |
| MULTILINE (8) | Matches can cross line boundaries |
| LITERAL (16) | Ignore escape characters and metacharacters in the pattern |
| DOTALL (32) | Enables the dot metacharacter to match line terminator characters also |
| UNICODE_CASE (64) | Enables Unicode case insensitive matching |
| CANON_EQU (128) | Matches Unicode characters only if their full canonical decomposiitons match (see the Unicode standard) |

  - By default, all of the flags are set off; they can be set on using the second form of the compile() method: the second operand specifies the flag values to use

- There are many more intricacies to regular expressions, and the person who needs more capabilities than discussed here is directed to the API docs web pages

# Using Regular Expressions

- Once you have constructed the regular expression you are now ready to apply the expression against some data; again, the steps:
  - Create a Pattern instance by compiling your regular expression:

```
Pattern pno = Pattern.compile("reg_exp")"
```

  - Create a Matcher instance by invoking the matcher() method of Pattern on your pattern instance, against the data to check:

```
Matcher mtch = pno.matcher(input_data);
```

  - Check if the input data matches the pattern by invoking the matches() method against your matcher:

```
if ( mtch.matches() ) { ... }
```

- These techniques should allow you to validate most incoming String data
  - But there's one more step we can take ...

# Data Validation in DataManager

- **Another aspect of data validation we can add is to handle non-numeric data entered for the** int **and** float **fields**
  - **Currently, we just issue a message and return the default value of -1 to signify the user is cancelling the request**
  - **Instead, we could issue a message at the point of data entry (in DataManager itself) and stay in a loop until the user enters a valid numeric value or explicitly cancels the request, for example...**

**In PromptandGetInt:**

```
public  static int  PromptAndGetInt(String text, BufferedReader in2)
    {
String stringIn = null;
int backInt = -1;
boolean goodInt = false;

while (!goodInt) {
 System.out.print("Please enter the value you want for "
     + text + ", or 'c' to cancel: ===> ");

   try { stringIn = in2.readLine();
         if (stringIn.equals("c")) {
                 goodInt = true;
                 backInt = -1;       }
         else { backInt = Integer.parseInt(stringIn);
                 goodInt = true;
               }
        }
   catch (IOException ex)
       {
        System.out.println("In IOException routine.");
       }

   catch (NumberFormatException ex)
       {
        System.out.println(stringIn +
          " is not a valid integer. Please try again.");
       }
  }  // end of while loop

   return backInt;

}  // end of prompt-and-get for int field
```

# Data Validation in DataManager, 2

## In PromptandGetFloat:

```
public  static float PromptAndGetFloat(String text,
                                                    BufferedReader in2)
    {
String stringIn = null;
float backFloat = -1;
boolean goodFloat = false;

while (!goodFloat) {
 System.out.print("Please enter the value you want for "
     + text + ", or 'c' to cancel: ===> ");

    try { stringIn = in2.readLine();
          if (stringIn.equals("c")) {
                   goodFloat = true;
                   backFloat = -1;        }
          else { backFloat = Float.parseFloat(stringIn);
                   goodFloat = true;
                }
         }

    catch (IOException ex)
        {
         System.out.println("In IOException routine.");
        }

    catch (NumberFormatException ex)
        {
         System.out.println(stringIn +
               " is not a valid number. Please try again.");
        }
   }  // end of while loop

    return backFloat;

}  // end of prompt-and-get for float field
```

Computer exercise: Object Data Validation

Set up your data entry application (InventoryDataManager) to validate the incoming data, based on the lecture content:

1. Modify InventoryDataManager.java by deleting the entire getFields() method and in its place copy in the file named DataValidate: this contains the code needed to implement our data validation logic. Recall that Pattern and Matcher are classes in java.util.regex; you will need to add an import statement to import all classes in this package.

2. Compile InventoryDataManager.java.

3. Copy DM3 from your J510 directory into your J510 directory as DataManager.java, totally replacing the current version:

   cp DM3 DataManager.java

4. Compile DataManager.java

5. Test InventoryDataManager; in particular, be sure to test:

   * Entering bad data is detected and prevented
   * starting a new entry but cancelling it at various points still works: that is, test the cancel option for each of unit price, quantity on hand, description, and part number.

**Optional exercise stretches:**

Experiment with alternative ways of building the regular expression we used in testing.

Note that we have not really done anything about validating the expiration date for Foodstuffs objects. That will come later.

# Section Preview

- **The Case Study Part 7**
  - **Overwriting Files**
  - **Objects and Files**
  - **Preventing Overwriting of Files (Machine Exercise)**

# Overwriting Files

❒ **There is one more issue regarding creating files we need to address before moving on: suppose there is already a file with the same name as the one we have just entered?**

- ♦ **Under our current logic, we would simply replace it, perhaps losing important data**
  - ✗ We probably need to be a little more careful

❒ **The right approach is something like this:**

- ♦ **After we have ensured the part number is a valid format, we ouught to check if a file by that name already exists**
  - ✗ We should do this even before we prompt for the other fields
- ♦ **If no such file already exists, we can move on, but if there already is such a file, we should alert the user and give him these choices:**
  - ➢ Continue, and let the new file replace the existing one (just press <Enter>, say)
  - ➢ Cancel this entry (just enter a 'c')
  - ➢ Simply enter a different part number and stay in the validation logic

# Overwriting Files, 2

❒ **Implementing this plan in code ...**

**Checking if a file exists**

♦ **We have a file name; to see if a file by that name exists, we use the File class and its exists() method; the parts:**

```
File testFile;   // declare a File variable
.
.
.
testFile = new File(workPartno);
if (testFile.exists())
```

✗ At this point, we need to make clear that instantiating testFile as a new File does not, in and of itself, create an actual file - it creates an instance of a File object

➢ Think of this as simply a collection of fields (attributes) with various values, nothing permanent yet

➤ Thus the **if** statement simply asks the question if there is a real file with the same name as the value in workPartno

# Overwriting Files, 3

❒ **Implementing this plan in code ...**

**Alert the user and give him three choices**

```
  System.out.println("A file with the name " +
                workPartno + " already exists.");
  System.out.println("Press <Enter> if you wish " +
               " to replace the existing file,");
  System.out.println("Enter a new part number if " +
               "you wish to change the name, or");
  System.out.println("enter 'c' if you wish to " +
               "cancel adding this inventory item.");
```

**Get the user's response**

```
try   { lineIn = rdr2.readLine(); }

catch  (IOException ex)
      { System.out.println("In IOException routine."); }
```

**Analyze and act on the user's response**

```
  finally   {
       if (lineIn.length()==0)  { validPno = true; }
       else {
              if (lineIn.equalsIgnoreCase("c"))
                 { writeOK = false; }
              else   { workPartno = lineIn;
                       writeOK = true; }
             }
            }
  }   // end of if from previous page
```

# Objects and Files

- The whole point of these last several sections has been an exploration of how to make objects persistent: to last beyond the run of the program where they are created

- This is done by saving the attribute values for each such object on an external store, usually disk (DASD, as we say in the mainframe world)

- Since the "native" encoding of the JVM is UTF-16, we have decided to store character and string data in UTF-16 and numeric data in IEEE native formats

- In later programs, when a user needs to work with an existing object, we will provide a constructor in the class to retrieve the data from external storage and use these values to initialize the newly instantiated object

- The actual data on DASD may be in the form of an HFS file, an MVS file, or some external database

  - Discussion of Java and databases is beyond the scope of this course

# Objects and Files, 2

- In addtion, Java programs may need to access data from external storage that is not related to any Java objects at all
  - In our case study, we will eventually be processing a flat file transaction log, for example
- While data meant to be shared across Java programs should normally be encoded in UTF-16 and native numeric formats, data meant to be shared between Java applications and non-Java applications should normally be encoded in the native format(s) of the host machine
  - Such data is, of course, not directly transferable across all Java-supported platforms
  - And some data is not valid in non-Unicode encodings
- So you need to plan carefully how data will be used
  - And you need to document clearly the intended usage, actual encodings, thoughts for moving code and data to other platforms, and the like

# Objects and Files, 3

- **So at this point, we have two classes (InventoryDataManager and DataManager) that work together to allow a user at a JVM terminal to add objects that are persistent**
  - **The objects are stored, each in their own HFS file, encoded in UTF-16 and IEEE numeric formats**
  - **Further, the data are validated as follows**
    - ✗ File name (that is, the part number) is of the form PARTddddd or FOODddddd
    - ✗ A file of the specified name does not already exist
    - ✗ The quantity on hand is an integer in the correct range
    - ✗ The unit price is a floating point number in the correct range
  - **In addition, the process allows the user to cancel or change their mind at any step along the way**

Computer exercise: Preventing Overwriting of Files

1. Update InventoryDataManager.java to add code to prevent overwriting any existing files, as follows:

   In the method getFields(), when you test for a valid pattern ( **if (m.matches))** ) is this following code:

```
{
// if get here, a valid part number
validPno = true;
}  // done handling valid part no
```

   replace the line validPno = true; with the file IDM5.

   Also in getFields(), above the while loop, add these declares:

```
String lineIn = null;
File  testFile;
```

2. Compile and test InventoryDataManager.java, especially test attempting adding a new file with a part number that matches an existing file name.

# Section Preview

- **Reading from HFS files**
  - **Reading fields and records**
  - **Classes for file input**
  - **InputStream classes for file input**
  - **Reclaiming data**
  - **Instantiating Objects**
  - **Using the FileInputStream Class for File Input**
  - **Using the FilterInputStream Class for File Input**
  - **Using the BufferedInputStream Class for File Input**
  - **Using the DataInputStream Class for File Input**
  - **Reader classes for file input**
  - **Using the BufferedReader Class for File Input**
  - **Using the InputStreamReader Class for File Input**
  - **Using the FileReader Class for File Input**
  - **A Lesson in Scope**
  - **Add, display, update, and delete files (Machine Exercise)**

# Reading Fields and Records

- Now that we have data on disk, we need to figure out how to retrieve it - and thus we will have ways to instantiate objects whose field values were saved to disk so they will be non-transient

- In order to read data from a disk file, you need to know how it was written: field size(s), encoding, presence or absence of BOMs, and so on

- In some cases you must read data a field or "chunk" at a time
  - For example, if there are embedded BOMs
  - Also, if you have mixed data types (character string, int, and float, for example, as in our case study)

- Other times you may be able read a whole record then parse out fields to separate variables
  - We'll discuss / demonstrate both approaches

- Our examples for reading our files will try to use the classes analogous to the classes used for writing the files
  - Although in some cases there may not be an exactly analogous class for input
  - And often it is possible to use different classes

# Classes for File Input

❒ **When talking about streams that will read from HFS files, we have these classes available:**

- ♦ **InputStream - the abstract class that is the parent of all byte stream input classes**
  - ✗ FilterInputStream - reads or skips bytes; extends InputStream
    - ➢ BufferedInputStream - supports buffering; extends FilterInputStream
    - ➢ DataInputStream - supports reading Java primitives; extends FilterInputStream
  - ✗ FileInputStream - a "convenience class" for reading byte files; extends InputStream
- ♦ **Reader - the abstract class that is the parent of all character stream input classes**
  - ✗ BufferedReader - supports buffering; extends Reader
  - ✗ InputStreamReader - bridge from character streams to byte streams; supports encoding; extends Reader
    - ➢ FileReader - a "convenience class" for reading character files; extends InputStreamReader

❒ **Note there are no Input classes analogous to the PrintStream and PrintWriter Output classes**

# Classes for File Input, 2

❒ **How do you choose, how do you use, these classes?**

- ♦ **On the following pages we discuss the most useful classes listed on the previous page, including showing samples of code and describing the results**
  - ✗ While extensive, it is not exhaustive, in that we haven't talked about all methods for all classes - but we provide a framework and some solid background for further exploration on your own

❒ **For the InputStream classes, the constructors generally need a [first] argument of "an input stream"**

- ♦ **This is fulfilled by an instantiation of FileInputStream:**

  **new FileInputStream(***filename***)**

  **produces an input stream that will read from the file named** *filename*

❒ **For the Reader classes, the constructors generally need a [first] argument of "a reader"**

- ♦ **This is fulfilled by an instantiation of Reader:**

  **new Reader(***filename***)**

  **produces a Reader stream that will read from the file named** *filename*

# InputStream Classes for File Input

- **We start by examining InputStream classes**

- **Here is a short summary of the methods available in the parent (abstract) class; all the subclasses inherit or implement these methods**

**InputStream methods**

- **available() - returns an integer containing an estimate of the number of bytes that can be read or skipped without blocking by the next invocation of a method for this stream**

- **close() - frees the resources associated with this stream**

- **mark(*readlimit*) - make note of current position in this stream; if *readlimit* bytes are subsequently read, the mark location may not be remembered longer**

- **markSupported() - returns boolean** true **or** false **indicating if the methods** mark **and** reset **are supported**

- **read() - reads one byte of data; blocks until data is available, end of stream is detected, or an exception is thrown; returns an int in the range of 0-255; if end of stream detected, returns -1**

- **read(*byte_array*) - read from the stream into the byte array; returns an integer containing the number of bytes read, never more than the size of the array; returns -1 if end of stream detected**

# InputStream Classes for File Input, 2

- **read(***byte_array***, *strt*, *len*) - read from the stream into the byte array (begin at the *strt*-th entry, for a mximum of *len* bytes); returns actual number of bytes read, -1 if none read due to end of stream being detected**

- **reset() - reposition to the last** mark**-ed location**

- **skip(***num_bytes***) - skip** *num_byte* **bytes;** *num_byte* **is a long; returns acutal number of bytes skipped as a long**

# Reclaiming Data

- **We can easily retrieve byes from a file, unchanged, into a variable**
  - **This is demonstrated repeatedly in the following pages**
- **But to actually work with data, we may need to perform various transformations:**
  - **Character data read in as bytes will have to be converted to String, and may need to be encoded differently, for example:**

```
FileIn.read(inBytes,0,9);
workString = new String(inBytes,0,9,"iso-8859-1");
```

  - **Integer data, if read in as bytes and converted to String, has to be converted to** int**, for example:**

```
workQOH = Integer.parseInt(workString);
```

  - **Float data, if read in as bytes and converted to String has to be converted to** float**, for example:**

```
workUnPr = Float.parseFloat(workString);
```

- **Recall that in some cases we stored integer and floating point data in native format, so we need to read them in that way, then there is no conversion to do**

# Instantiating Objects

- **After we read a file and obtain the field values, we still don't have an object**

- **Generally, we will read the fields from a file into work variables, as a stream of bytes or characters:**

```
String workPartNo;
String workDesc;
int    workQOH;
float  workUnPr;
```

  - **And we can instantiate an object with dummy / default values to use in our work:**

```
InventoryItem  workItem =
        new InventoryItem(" ", " ", 0, 0.00f);
```

  - **After we extract the values from a file, and do any necessary conversion, we need to populate workItem, using InventoryItem's** set **methods**

  - **And then do whatever work is required on the object**

    - ✗ For purposes of illustration, we will simply display the fields in the current object, for example

```
System.out.print("Part number = " + workItem.getPno());
System.out.print(", QOH = " + workItem.getQOH());
System.out.print(", Unit pr. = " + workItem.getUnPr());
System.out.println(", Desc = " + workItem.getDesc());
```

# Using the FileInputStream Class for File Input

- **The FileInputStream class extends InputStream and implements its methods**

  **Constructors**

  - **FileInputStream(***File_object***)**
  - **FileInputStream(***file_name***)**
  - **FileInputStream(***FileDescriptor_object***)**

  **Example**

```
String filename = "PART005005";
FileInputStream FileRecIn =
                new FileInputStream(filename);
```

- **Note that we can use our part numbers as filenames, then open the file by instantiating the related stream**
  - **Clearly, reading the file, by field or by record, will allow us to assign the saved values to the fields in an object**

- **Also note that the first two constructors might throw a FileNotFoundException, and all three might throw a SecurityException if access is not permitted**

# Using the FileInputStream Class for File Input, 2

❒ **The files we created using the FileOutputStream class were these:**

**PART00505 - ASCII (code page ISO-8859-1)**
**PART00506 - EBCDIC (CP037)**
**PART00507 - UTF16 (UTF16)**

❒ **To read these requires using one of the** read **methods, which, for this class, are all byte streams related (assuming we do not want to read just one byte at a time)**

- **These methods require a buffer that is a byte array, which at first glance might be something like this:**

```
byte[] inBytes;
```

- **But attempting to use inBytes as an argument to a** read **method returns a warning message from javac that the array might be uninitialized!**

  ✗ This is surprising the first time you see it: who cares what the initial value is, since I'll be reading into it anyway

  ✗ But what it's about is Java doesn't know how much memory to allocate; so we supply an allocation large enough to hold more than enough data:

```
inBytes = new byte[200];
```

  ✗ ... and we're good to go for all our byte stream methods

# Using the FileInputStream Class for File Input, 3

❒ **The** read **methods all return an int (-1 for end of file, otherwise the number of bytes returned), so we define**

```
int noBytes;
```

❒ **To access the fields from PART00505 (ASCII):**

```
noBytes = FileRecIn.read(inBytes,0,9);
noBytes = FileRecIn.read(inBytes,0,30);
noBytes = FileRecIn.read(inBytes,0,9);
noBytes = FileRecIn.read(inBytes,0,10);
```

♦ **After each read, you have the bytes from the file, in their original encoding, in the inBytes buffer**

✗ Between the reads, you need to save the data into the appropriate variables (convert byte array to String), and do any additional necessary conversions, then set the field values; for example:

```
noBytes = FileRecIn.read(inBytes,0,9);
workPartNo = String(inBytes,0,9,"ISO-8859-1");
workItem.setPno(workPartNo);

noBytes = FileRecIn.read(inBytes,0,30);
workDesc = String(inBytes,0,30,"ISO-8859-1");
workItem.setDesc(workDesc);

noBytes = FileRecIn.read(inBytes,0,9);
workString = new String(inBytes,0,9,"ISO-8859-1");
workQOH = Integer.parseInt(workString);
workItem.setQOH(workQOH);

noBytes = FileRecIn.read(inBytes,0,10);
workString = new String(inBytes,0,10,"ISO-8859-1");
workUnPr = Float.parseFloat(workString);
workItem.setUnPr(workUnpr);
```

# Using the FileInputStream Class for File Input, 4

❒ **To access the fields from PART00506 (EBCDIC), use the same logic as on the previous page (although we could omit the code pages, since that's the default for z/OS)**

❒ **Here's some code to read the entire file as a single record and then extract the fields from that:**

```
filename = "PART00506";
 FileInputStream FileRecIn =
                    new FileInputStream(filename);
 noBytes = FileRecIn.read(inBytes,0,58);
 workItem.setPno(new String(inByres,0,9));
 workItem.setDesc(new String(inBytes,9,30));
 workItem.setQOH(Integer.parseInt(new String(inBytes,39,9));
 workItem.setUnPr(Float.parseFloat(new String(inBytes,48,10));
 FileRecIn.close();
```

- **Notice, too, how in this example we nested method calls and avoided using the intermediate (work) variables**

  ✗ This is just for show; it's probably easier to understand and maintain using the intermediate variables

❒ **All of the work in this section is done in the context of** try, catch, **and** finally **blocks, of course**

# Using the FileInputStream Class for File Input, 5

- To access the fields from PART00507 (UTF-16):

- Recall that when we wrote out all data as UTF-16 fields, each field ended up with a Byte Order Mark (BOM) in front of it - we need to skip each of these (2 bytes each)

  - Also, each field is twice as large as for ASCII and EBCDIC, so our code looks something like this:

```
noBytes = FileRecIn.read(inBytes,0,2); // BOM
noBytes = FileRecIn.read(inBytes,0,18);
workPartNo = String(inBytes,0,18,"utf-16");
workItem.setPno(workPartNo);

noBytes = FileRecIn.read(inBytes,0,2); // BOM
noBytes = FileRecIn.read(inBytes,0,60);
workDesc = String(inBytes,0,60,"utf-16");
wokrItem.setDesc(workDesc);

noBytes = FileRecIn.read(inBytes,0,2);  // BOM
noBytes = FileRecIn.read(inBytes,0,18);
workString = new String(inBytes,0,18,"utf-16");
workQOH = Integer.parseInt(workString);
workItem.setQOH(workQOH)l

noBytes = FileRecIn.read(inBytes,0,2);  // BOM
noBytes = FileRecIn.read(inBytes,0,20);
workString = new String(inBytes,0,20,"utf-16");
workUnPr = Float.parseFloat(workString);
workItem.setUnPr(workUnPr);
```

# Using the FilterInputStream Class for File Input

- **The FilterInputStream, unlike the FilterOutputStream, only has one constructor, and it is marked** protected
  - **This means the constructor is accessible only from methods in classes in the same package**
  - **So we really can't use this to access files directly**
    - ✗ We used its subclass BufferedInputStream to read files created using FilterOutputStream

# Using the BufferedInputStream Class for File Input

## Constructors

- **BufferedInputStream(***input_stream***)**
- **BufferedInputStream(***input_stream***, *buf_size*)**

## Example

```
filename = "PART00205";
BufferedInputStream bis1 = new BufferedInputStream(
                  new FileInputStream(filename), 200);
```

## Methods

- **The methods are all the same as for FileInputStream, presumably with better performance due to buffereing**

❒ **Techniques and results are the same as with FileInputStream class**

# Using the DataInputStream Class for File Input

- **Since we finally decided to use DataOutputStream for our case study, and to record data in UTF16 for string and native internal format for numeric items, this will class will be most useful for us**

**Constructor**

- **DataInputStream(***input_stream***)**

**Example**

```
filename = "PART00305";
DataInputStream RecIn =
    new DataInputStream(new FileInputStream(filename));
```

- **Now, as with DataOutputStream, we have some different methods available ...**

# Using the DataInputStream Class for File Input, 2

- **The methods for DataInputStream include the methods inherited from the FilterInputStream, but also some additional methods:**
  - **readBoolean() - read one byte as a boolean true or false**
  - **readByte() - read one byte as a byte type**
  - **readChar() - read one Unicode character**
  - **readDouble() - read 8 bytes, interpreting as double**
  - **readFloat() - read 4 bytes, interpreting as float**
  - **readFully(*byte_array*) - read bytes into the specified array until it is full or end of data is encountered**
  - **readFully(*byte_array*, *strt*, *len*) - read bytes into the specified array, beginning at offset *strt*, until *len* bytes have been read, end of array is encountered, or end of data is encountered**
  - **readInt() - read 4 bytes, interpreting as an integer**
  - **readShort() - read 2 bytes, interpreting as a short**
  - **readUnsignedByte() - read one byte, interpreting as unsigned byte**
  - **readUnsignedShort() - read two bytes, interpreting as unsigned short**
  - **readUTF() - read a string that is prefixed by an unsigned short that contains the length of the remaining UTF-8 bytes; convert this string to UTF-16 String data**
  - **readUTF(*input_stream*) - read in string encoded as UTF-8 and convert to UTF-16 String data; this is a static (class) method**

# Using the DataInputStream Class for File Input, 3

- The code we use to read our UTF-16 files, then, looks familiar, but simpler than before since we have no embedded BOMs to consider:

```
noBytes = RecIn.read(inBytes,0,18);
workPartNo = String(inBytes,0,18,"utf-16");
workItem.setPno(workPartNo);

noBytes = RecIn.read(inBytes,0,60);
workDesc = String(inBytes,0,60,"utf-16");
workItem.setDesc(workDesc);

workQOH = RecIn.readInt();
workItem.setQOH(workQOH);

workUnPr = RecIn.readFloat();
workItem.setUnPr(workUnPr);
```

  - Also note this is an example where int and float data were written out in native format, so no conversion is required for these items

# Reader Classes for File Input

❒ **Here we look at Reader classes for reading HFS files; generally speaking, Readers work with characters as opposed to bytes, so for an input buffer we use:**

```
char[] inChars;
inChars = new char[100];
```

❒ **Here is a short summary of the methods available to all Reader classes:**

**Reader methods**

- **close() - frees the resources associated with this stream**
- **mark(*readlimit*) - make note of current position in this stream; if *readlimit* bytes are subsequently read, the mark location may not be remembered longer**
- **markSupported() - returns boolean** true **or** false **indicating if the methods** mark **and** reset **are supported**
- **read() - reads one character; blocks until data is available, end of stream is detected, or an exception is thrown; returns a character as an int in the range of 0-65535; if end of stream detected, returns -1**
- **read(*char_array*) - read from the stream into a character array; returns an integer containing the number of characters read, never more than the size of the array; returns -1 if end of stream detected**

# Reader Classes for File Input, 2

❒ **A short summary of the methods available to all Reader classes, continued:**

- **read(*char_array*, *strt*, *len*) - read from the stream into a character array (begin at the *strt*-th entry, for a mximum of *len* characters); returns actual number of character read, -1 if none read due to end of stream being detected [note: this is an abstract method]**

- **read(*char_buffer*) - read from stream into the specified character buffer, unchanged; returns number of characters added to buffer or -1 if source is at end of file**

- **ready() - boolean; returns true if stream is ready to be read**

- **reset() - reposition to the last** mark**-ed location**

- **skip(*num*) - skip *num* characters; *num*is a long; returns acutal number of characters skipped as a long**

# Using the BufferedReader Class for File Input

❒ **The BufferedReader class extends Reader and implements its methods**

**Constructors**

- **BufferedReader(***reader***)**
- **BufferedReader(***reader***, *buf_size*)**

**Examples**

```
filename = queryPartNo;
BufferedReader bfrdr =
  new BufferedReader(new FileReader(filename));


filename = "PART01205";
BufferedReader br1 =
  new BufferedReader(new FileReader(filename), 100);
```

- **The docs say the default buffer size is generally fine, but the second example shows allocating a buffer size of your choice**

# Using the BufferedReader Class for File Input, 2

❒ **The methods available for this class are the same ones as for the parent class, plus this additional method:**

- ♦ **readLine() - returns a String from the reader that includes all characters up to, but not including, any of the line delimiters (CR, LF, CRLF)**

  ✗ Since our files are not delimited this way, we'll not use this method here

❒ **Then, in the code example below, we read and assign the data contained in our file to the various fields, populate workItem, then close the BufferedReader:**

```
  noChars = br1.read(inChars,0,9);
  workItem.setPno(new String(inChars,0,9));
  noChars = br1.read(inChars,0,30);
  workItem.setDesc(new String(inChars,0,30));
  noChars = br1.read(inChars,0,9);
  workItem.setQOH(Integer.parseInt(new
                              String(inChars,0,9)));
  noChars = br1.read(inChars,0,10);
  workItem.setUnPr(Float.parseFloat(new
                              String(inChars,0,10)));

br1.close();
```

# Using the InputStreamReader Class for File Input

- **The InputStreamReader class extends Reader and implements its methods**

  **Constructors**

  - **InputStreamReader(***reader***)**
  - **InputStreamReader(***reader***, *character_set*)**
  - **InputStreamReader(***reader***, *character_set_decoder*)**
  - **InputStreamReader(***reader***, *character_set_name*)**

- **If no character set object, decoder, or name is specified, the default character set for the underlying platform is assumed**

  **Example**

```
filename = "PART01305";
InputStreamReader isr1 =
    new InputStreamReader(new FileInputStream(filename));
```

# Using the InputStreamReader Class for File Input, 2

- The InputStreamReader class has the methods we've already discussed for Reader classes, plus:
  - getEncoding() - returns a String containing the name of the character encoding for the stream
- So the code for using this class is the same as we used for the BufferedReader class, at least for our needs

# Using the FileReader Class for File Input

- **The FileReader class extends InputStreamReader and implements its methods**

  **Constructors**

  - **FileReader(***file_object***)**
  - **FileReader(***file_descriptor***)**
  - **FileReader(***filename***)**

  **Example**

```
filename = "PART01505";
FileReader fr1 = new FileReader(new File(filename));
```

- **And the methods / code to use are the same as for InputStreamReader**

# A Lesson in Scope

- **To avoid the tedious process of creating an interesting amount of data by hand, we developed a Java program to create 100 PART*nnnnn* files automatically**
  - **And in the process of doing this, we learned some practical lessons, which we discuss here**

- **First, we created a file called "Descriptions", based on lab data from a different course, with these properties:**
  - **100 records, each 60 bytes long**
  - **Each record contains 30 UTF-16 characters that will be used for an inventory item's description field**
  - **Records 11 and 99 will have UTF-16 characters that are non-Latin**

# A Lesson in Scope, 2

❒ **Next we created our Java program, named createItems.java, with this basic logic:**

- ♦ **Set an integer, workCntr, to be 8000**
- ♦ **Open Descriptions**
- ♦ **For each record in Descriptions,**
  - ✗ Create a file name of PART0 + workCntr.toString(); use this as the file name and the part number
  - ✗ Add the content from the current Descriptions record as the item description field
  - ✗ Use the Java random number generator to create values for the quantity on hand and unit price
  - ✗ Write the fields out and close the output file
  - ✗ Increment workCntr

❒ **The idea was clear, but we encountered some snags along the way ...**

# A Lesson in Scope, 3

❒ **The first problem was in declaring our input file; we started with:**

```
            DataInputStream inDescs =
new DataInputStream(new FileInputStream("Descriptions"));
```

♦ **Which raised this compile error:**

**unreported exception java.io.FileNotFoundException;**
**must be caught or declared to be thrown**

❒ **Ah! Yes. Of course, there is some theoretical possibility the input file does not exist**

♦ **So, the declare must be done in the context of** try **and** catch **blocks, perhaps like this:**

```
try {
  DataInputStream inDescs =
new DataInputStream(new FileInputStream("Descriptions"));
     }

catch (FileNotFoundException ex)
 {
   System.out.println("Descriptions was not found.");
 }
```

♦ **Well, this compiles; but when we try to read the file in a loop...**

# A Lesson in Scope, 4

```
try {
   DataInputStream inDescs =
new DataInputStream(new FileInputStream("Descriptions"));
     }

catch (FileNotFoundException ex)
 {
    System.out.println("Descriptions was not found.");
 }

 do {
    try
     {
      noBytes =  inDescs.read(inBytes,0,60);
     }
    catch (IOException ex)
     {
      System.out.println("In IOException for read.");
     }
       .
       .
       .
     }  while(!eof);
```

- **We're knowledgable enough to put the** read **in a** try **and** catch **block context, but now we get this compile time error:**

```
createItems.java:40: cannot find symbol
symbol   : variable inDescs
location: class createItems
            noBytes =  inDescs.read(inBytes,0,60);
                       ¬
```

❒ **And it was this error that took us two days to figure out**

# A Lesson in Scope, 5

- **Because the stream** inDescs **is declared and instantiated inside a** try **block, it is not known inside a separate** do **block, even though they are in the same method!**

- **The solution is to declare inDescs outside of the try block, so it is known everywhere in the code, something like this:**

```
 DataInputStream inDescs = null;

try {
   inDescs =
new DataInputStream(new FileInputStream("Descriptions"));
     }

catch (FileNotFoundException ex)
 {
    System.out.println("Descriptions was not found.");
 }

 do {
    try
     {
      noBytes =  inDescs.read(inBytes,0,60);
     }
    catch (IOException ex)
     {
      System.out.println("In IOException for read.");
     }
       .
       .
       .
     }  while(!eof);
```

- **And this solved the most intractable problem; it just seemed worthwhile to demo the problem and its solution here**

Computer Exercise: Add, Display, Update, and Delete Files

This exercise combines, consolidates, and ties together many parts of Java and file work that we have been discussing. This is a large, complex lab, with many opportunities to learn.

First, in your J510 directory is a program called **testIO_inStreams.java**. This program demonstrates using various classes and methods for reading the files we created earlier. It is not essential for our work, but you might like to have sample code for later work of your own. **Optionally,** compile and run this program.

Next, the program, **createItems.java**, discussed on pages 402-406 is available in your J510 directory. Compile and run this program. This will create the 100 PART08*nnn* files we talked about. These files can be used in testing our next version of InventoryDataManager; then, you can always re-run the createItems program to reset to a common starting point.

**You might find it instructive to read the code, especially the part where we use the random number generator to generate quantity on hand and unit price.**

Thirdly, modify **InventoryDataManager.java** to add support for option 3 (Display data in HFS), as follows:

a. At the top of code, before your main(), add a declare and instantiation of workItem as a working object, like this:

```
static InventoryItem workItem =
                new InventoryItem(" "," ",0,0.00f);
```

b. In your menu-processing logic, replace your code for case 3 with the contents of the file called **case3** in your J510 directory

c. After the end of your main method (but before the end of the class definition), copy in file **display** from your J510 directory; this contains methods **displayItem()** and **GetFileName(*request_type*)**.

d. Compile and test; at this point you can add new objects and display any existing object created using our chosen methods (use the items with part numbers 8000 or higher for your testing).

Next, modify **InventoryDataManager.java** to add support for option 7 (Delete data in HFS), as follows:

a. In your menu-processing logic, replace your code for case 7 with the contents of the file called **case7** in your J510 directory

b. After the end of your main method (but before the end of the class definition), copy in file **delete** from your J510 directory; this contains the method **deleteItem()**

c. Compile and test; at this point you can add new objects and display and delete existing object created using our chosen methods (use the items with part numbers 8000 or higher for your testing).

Finally, modify **InventoryDataManager.java** to add support for option 5 (Update data in HFS), as follows:

a. In your menu-processing logic, replace your code for case 5 with the contents of the file called **case5** in your J510 directory

b. After the end of your main method (but before the end of the class definition), copy in file **update** from your J510 directory; this contains the methods **updateItem()** and **getSubFields()**

c. Compile and test; at this point you can add new objects and display, update, and delete existing object created using our chosen methods (use the items with part numbers 8000 or higher for your testing).

# Section Preview

- **Numbers and the java.math Package**
  - **Looking at Unit Price**
  - **Precision**
  - **Rounding**
  - **The MathContext class**
  - **The BigDecimal class**
  - **The BigInteger class**
  - **The java.math package**
  - **Using BigDecimal (Machine exercise)**

# Looking at Unit Price and Other Floating Point Values

❒ **After the last couple of labs, we have a solid number of instances of InventoryItem to work with, and little line command utility (InventoryDataManager) to create, examine, change, and delete the files that contain instance data**

❒ **If you look at some of the items we have, using our utility, you might see something like this:**

```
Part number = PART08000, QOH = 3, Unit price = 10.70659...
Part number = PART08003, QOH = 4, Unit price = 0.266408...
```

- **And so on; since we used a random number generator for both quantity on hand and unit price, your values may differ**

  ✗ But the value of most unit price fields will have a variable number of positions to the right of the decimal point when what we really want is two decimal places for all unit price values

  ➢ Note that the Japanese Yen, for example, is not subdivided, but most other world currencies maintain two decimal places for monetary amounts

# Looking at Unit Price and Other Floating Point Values, 2

❒ **There are other issues raised by floating point numbers, variables, and expressions**

**Consider:**

```
float f = .1f;
System.out.println(" f = " + f);
for (int i=1; i<8; i++)
 {
    f += .1f;
    System.out.println(" f = " + f);
 }
```

♦ **Running this yields:**

```
f = 0.1
f = 0.2
f = 0.3
f = 0.4
f = 0.5
f = 0.6
f = 0.70000005        <== yikes!
f = 0.8000001
```

✗ Based on an example in "Developing Mainframe Java Applications" by Lou Marco (Wiley)

# Looking at Unit Price and Other Floating Point Values, 3

❒ **Another example illustrates more issues**

**Consider:**

```
double gross = 0.70d;
double tax   = gross * .05;
System.out.println("\n Printing out gross: " + gross);
System.out.println("\n Printing out tax:   " + tax);
```

♦ **Running this yields:**

```
Printing out gross: 0.7
Printing out tax:   0.03499999999999996
```

♦ **But, of course, 0.70 * .05 should be 0.035**

✗ Based on an example in "Murach's JavaSE 6" by Joel Murcah and Andrea Steelman

❒ **So we need to examine some techniques for solving these issues**

❒ **We first review / present a little background in issues of precsion and rounding to form a basis for the next classes we will be introducing**

♦ **These values, in turn, will establish the current MathContext**

# Precision and RoundingMode

❒ **In the Java docs, we find these definitions:**

> **1. precision: the number of digits to be used for an operation; results are rounded to this precision**
>
> **2. roundingMode: a RoundingMode object which specifies the algorithm to be used for rounding.**

❒ **All arithemtic operations can be influenced by the setting for precision**

- ♦ **Digits outside the precision range can be truncated or rounded**

❒ **Rounding can occur in assignment and, typically in multiplication and division**

- ♦ **Java supports eight modes of rounding:**
  - ✗ Ceiling - round towards positive infinity
  - ✗ Down - round towards zero
  - ✗ Floor - round toward negative infinity
  - ✗ Half down - round towards "nearest neighbor", unless both neighbors are equidistant, then round down
  - ✗ Half even - round towards "nearest neighbor", unless both neighbors are equidistant, then round towards the even neighbor
  - ✗ Half up - round towards "nearest neighbor", unless both neighbors are equidistant, then round up
  - ✗ Unnecessary - asserts operation will not need rounding (if an inexact result occurs, an ArithmeticException is thrown)
  - ✗ Up - round away from zero

# Precision and RoundingMode, 2

- The Java API doc has a nice table in the discussion of the java.math package under RoundingMode to clarify by example
  - Suppose you have two-digit decimal values with the requirement to round to one digit (actually, the sample table has units.tenths values, with result if rounded to units):

| Input | ------ Result of rounding under various modes ------- | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Number | UP | DOWN | CEILING | FLOOR | HALF-UP | HALF-DOWN | HALF-EVEN | UNNEC. |
| 5.5 | 6 | 5 | 6 | 5 | 6 | 5 | 6 | * |
| 2.5 | 3 | 2 | 3 | 2 | 3 | 2 | 2 | * |
| 1.6 | 2 | 1 | 2 | 1 | 2 | 2 | 2 | * |
| 1.1 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | * |
| 1.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -1.0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1.1 | -2 | -1 | -1 | -2 | -1 | -1 | -1 | * |
| -1.6 | -2 | -1 | -1 | -2 | -2 | -2 | -2 | * |
| -2.5 | -3 | -2 | -2 | -3 | -3 | -2 | -2 | * |
| -5.5 | -6 | -5 | -5 | -6 | -6 | -5 | -6 | * |

  - You can find a deeper, more detailed discussion at http://java.sun.com/javase/6/docs/api/ then select java.math, then select RoundingMode
  - The default is HALF-UP, but you might have business cases where an alternative is preferred

# BigInteger and BigDecimal

- Recall when working with binary data, we have these choices of data types and wrapper classes:
  - byte and Byte - range of
    -127 to +127
  - short and Short - range of
    -32768 to + 32767
  - int and Integer - range of
    -214783658 to +2147483647
  - long and Long -range of
    -9223372036854775808 to +9223372036854775807
- All of these types are intended to be binary integer (that is, no fractional parts), and they each have a native hardware implementation on System z hardware
- The BigInteger class extends the abstract class Number and provides for a large number of methods and special operations
- The BigDecimal class also extends Number and provides additional functionality
  - Implemented on System z hardware using the decimal floating point formats and instructions

# BigInteger

❒ **The BitInteger class provides for arbitray precision integers**

- ♦ **The value for a BigInteger object is represented in as many bits as it takes (an "infinite word size abstraction")**

# Section Preview

❒ **Dates, Times, and Formatting**

- ♦
- ♦

# Section Preview

❒

- ♦
- ♦

# Section Preview

- **Appendices**
  - **Conversion rules**
  - **List of all Java-supplied Exception Classes**
  - **Unicode work**

This page intentionally left almost blank.

# Appendix A - Conversion rules

- Implicit conversions
- Explicit conversions
  - Casting
  - Methods

# Implicit Conversions

- **Conversions are implictyly applied to data in Java in these situations:**
  - **Data is displayed to the JVM console / terminal; numeric items are converted to string**
  - **Calculations are done with data items in a mix of formats**

# Explicit Conversions

- **Java can be told to convert data from one format to another explicitly by:**
  - **Using the cast operator**
  - **Invoking the appropriate method on an instance**

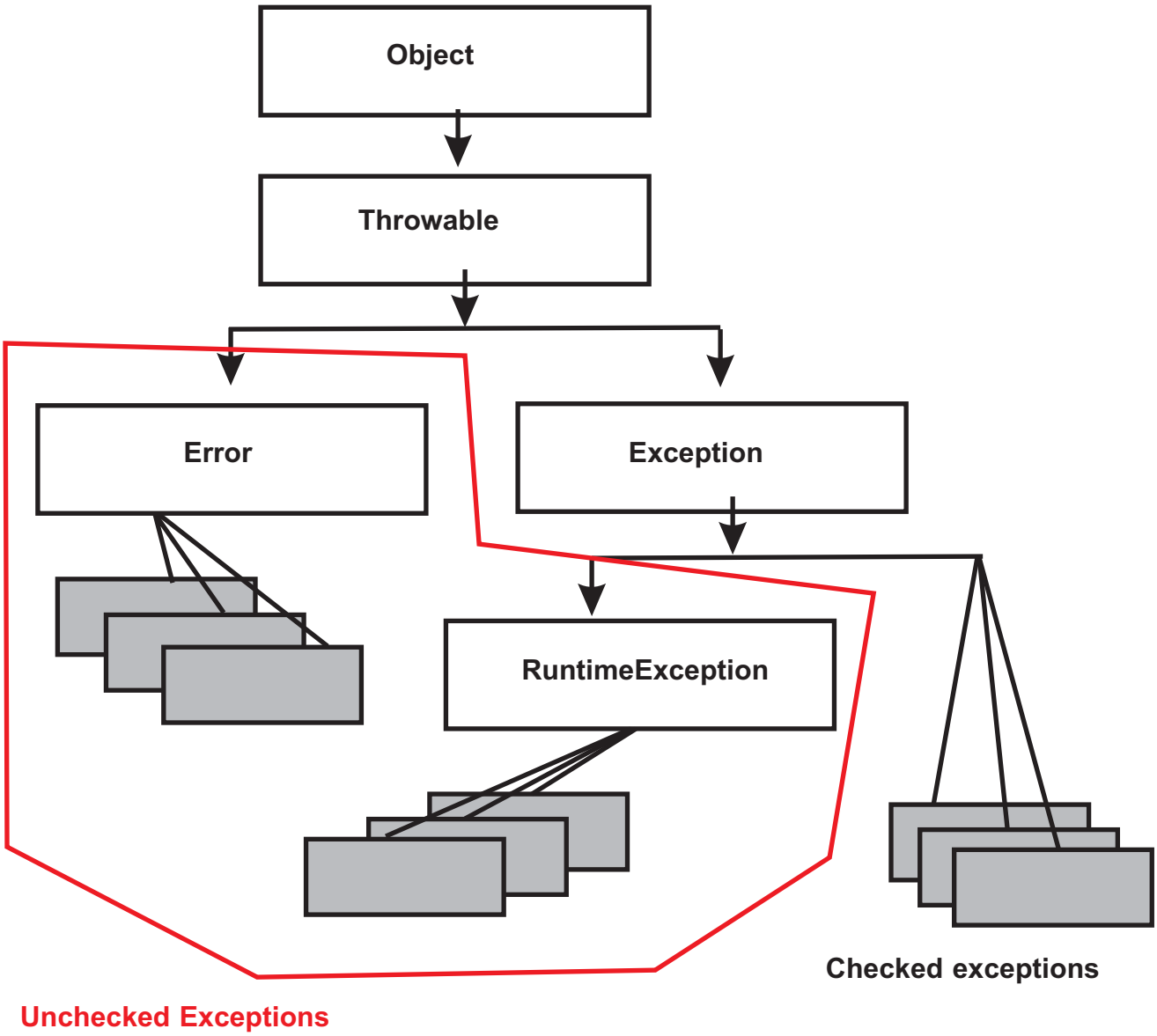
# Appendix B - List of all Java-supplied Exception Classes

❒ **The following is a simple list of all named exceptions found in the API docs**

- ♦ **Many are of no interest in this class, but this is just to give you an idea of the scope; some notes:**
  - ✗ CORBA - Commong Object Request Broker Architecture
  - ✗ AWT (Abstract Window Toolkit) and Swing relate to client-side programming
  - ✗ "Marshalling" is analogous to z/OS "allocation": ensuring necessary resources all available before proceeding
  - ✗ RMI - Remote Method Invocation

# Exception Hierarchy

- The starting class for exceptions is the Throwable class, which is a direct descendent of the Object class
  - Big picture:



# All Java Exceptions

- Following is a list of all Java exception classes documented in the Java SE version 6 API, including following the chain of "Known direct subclasses" - just under 500, with a brief description of each
  - Indentation shows subclassing
  - For more information, check the online documentation
  - We start with the Checked Exceptions:

# Checked Exceptions

```
AclNotFoundException                - reference made to unknown Access Control List
ActivationException                 - general error in activation interfaces
  UnknownGroupException                   - in RMI package
  UnknownObjectException                  - in RMI package
AlreadyBoundException               - attempt to bind an object that is already bound
ApplicationException                - ORB / CORBA problems
AWTException                        - Abstract Window Toolkit error
BackingStoreException               - preferences operation could not complete
BadAttributeValueExpException       - invalid MBean attribute passed to a query
BadBinaryOpValueExpException        - internal JMX query attribute passed
BadLocationException                - attempts to access non-existant parts of a document
BadStringOperationException         - invalid string passed to query
BrokenBarrierException              - attempt to wait upon a broken barrier
CertificateException                - one of below:
  CertificateEncodingException            - error attempting to encode certificate
  CertificateExpiredException             - self explanatory
  CertificateNotYetValidException         - self explanatory
  CertificateParsingException             - DER certificate errors
ClassNotFoundException              - self explanatory
CloneNotSupportedException          - attempt to use clone method but it's not supported
  ServerCloneException                    - remote exception occurs during cloning of a
                                              UnicastRemoteObject
DataFormatException                 - self explanatory
DatatypeConfigurationException      - serious configuration error
DestroyFailedException              - destroy [credential] attempt failed
```

# Checked Exceptions, continued

- ExecutionException - attempt to retrieve result from aborted task
- ExpandVetoException - stops expand / collapse of node
- FontFormatException - bad font file found
- GeneralSecurityException - one of below:
  - BadPaddingException - badly padded input for crypto
  - CertificateException - one of below:
    - CertificateEncodingException - attempt to encode failed
    - CertificateExpiredException - self explanatory
    - CertificateNotYetValidException - self explanatory
    - CertificateParsingException - DER certificate errors
  - CertPathBuilderException - failure in a certifcate path builder routine
  - CertPathValidatorException - self explanatiry
  - CertStoreException - problem retrieving certificate from a repository
  - CRLException - Certificate Revocation List error
  - DigestException - generic message digest exception
  - ExemptionMechanismException - generic exemption mechanism exception
  - IllegalBlockSizeException - data of incorrect length passed to cipher routine
  - InvalidAlgorithmParameterException - self explanatory
  - InvalidKeySpecException - invalid key specification
  - InvalidParameterSpecException - invalid parameter specification
  - KeyException - basic key exception
  - KeyStoreException - generic key store exception
  - LoginException - one of below:
    - AccountException - one of below:
      - AccountExpiredException - self explanatory
      - AccountLockedException - self explanatory
      - AccountNotFoundException - self explanatory
    - CredentialException - one of below:
      - CredentialExpiredException - self explanatory
      - CredentialNotFoundException - self explanatory
    - FailedLoginException - authentication failed
  - NoSuchAlgorithmException - crypto algorithim not found
  - NoSuchPaddingException - requested padding mechanism not available
  - NoSuchProviderException - requested security provider not available
  - ShortBufferException - output buffer too short
  - SignatureException - generic Signature exception
  - UnrecoverableEntryException - key store can not be recovered
    - UnrecoverableKeyException - key can not be recovered
- GSSException - GSS API error
- IllegalAccessException - method does not have access to array field or method
- IllegalClassFormatException - class file transformer error
- InstantiationException - class is not allowed to be instantiated

# Checked Exceptions, continued

InterruptedException - thread has been interrupted
IntrospectionException - not able to resolve or map a string or method
InvalidApplicationException - internal JMX exception; user code does not see this
InvalidMidiDataException - sound file not usable by this player
InvalidPreferencesFormatException - input did not conform to preferences XML doc type
InvalidTargetObjectTypeException - applies to MBeans
InvocationTargetException - thrown by an invoked method or constructor
IOException - one of below:
  ChangedCharSetException - charset has changed (Swing)
  CharacterCodingException - character encoding or decoding error
    MalformedInputException - character not legal for character set
    UnmappableCharacterException - input char. cannot be mapped to any output char
  CharConversionException - base for character conversion errors
  ClosedChannelException - I/O request to channel that is closed for that request
  EOFException - end of input stream
  FileLockInterruptionException - thread interrupted while waiting for file lock
  FileNotFoundException - self explanatory
  FilerException - filer detected an error
  HttpRetryException - HTTP request needs to be retried and cannot be retried automatically
  IIOException - run-time failure of read or write request
    IIOInvalidTreeException - run-time failure for IIOMetadata object (image i/o)
  InterruptedIOException - an I/O operation has been interrupted at thread level
    SocketTimeOutException - a timeout has occurred for socket read or accept
  InvalidPropertiesFormatException - invalid XML format
  JMXProviderException - provider for remote protocol is not usable
  JMXServerErrorException - remote MBean server error
  MalformedURLException - could not determine the protocol
  ObjectStreamException - superclass for all object stream exceptions
    InvalidClassException - class is invalid (no no-arg constructor, etc.)
    InvalidObjectException - object failed validation test
    NotActiveException - serialization or deserialization is not active
    NotSerializableException - no serializable interface exists but one required
    OptionalDataException - attempt to read object found primitive data type
    StreamCorruptedException - internal consistency checks violated
    WriteAbortedException - one of above exceptions occurred during a write
  ProtocolException - error in underlying protocol  such as TCP
  RemoteException - common supperclass for remote method calls
    AccessException - caller does not have permission for action
    ActivateFailedException - remote call fails to activate target
    ActivityCompletedException - further ongoing work is not possible
    ActivityRequiredException - mandatory activity not received
    ConnectException - connection refused by remote host

# Checked Exceptions, continued

| | |
|---|---|
| ConnectIOException | - I/O error during connection attempt |
| ExportException | - attempt to export a remote object has failed |
| InvalidActivityException | - request inconsistent with current context state |
| InvalidTransactionException | - request carried an invalid transaction context |
| MarshalException | - during a remote method call, marshalling of resources failed |
| NoSuchObjectException | - object no longer exists on the remote system |
| ServerError | - Error on the remote server |
| ServerException | - Exception on the remote server |
| ServerRuntimeException | - remote server running old Java release (1.1) |
| SkeletonMismatchException | - deprecated |
| SkeletonNotFoundException | - deprecated |
| StubNotFoundException | - valid stub class not found for remote object |
| TransactionRequiredException | - request carried null transaction |
| TransactionRolledbackException | - requested transaction has been rolled back or marked for rollback |
| UnexpectedException | - remote client recieved a checked exception not provided for |
| UnknownHostException | - while creating a connection |
| UnmarshalException | - variety of end-of-remote-transaction errors |
| SaslException | - Security error |
| AuthenticationException | - Security error |
| SocketException | - error in underlying protocol |
| BindException | - error binding socket to port |
| ConnectException | - could not connect |
| NoRouteToHostException | - intervening firewall, or router down |
| PortUnreachableException | - Port Unreachable message received |
| SSLException | - error detected by secure socket layer |
| SSLHandshakeException | - client and server could not negotiate desired level of security |
| SSLKeyException | - bad SSL key |
| SSLPeerUnverifiedException | - peer's identity not verified |
| SSLProtocolException | - flaw in protocol implementation |
| SyncFailedException | - sync operation has failed |
| UnknownHostException | - IP address of host could not be determined |
| UnknownServiceException | - bad MIME type or attempt to write to a read-only connection |
| UnsupportedDataTypeException | - requested operation does not support requested data type |
| UnsupportedEncodingException | - requested character encoding is not supported |
| UTFDataFormatException | - malformed UTF-8 string found |

# Checked Exceptions, continued

```
  ZipException                          - Zip exception of some sort has occurred
    JarException                            - error in working with JAR file
JAXBException                         - Java XML exception main class
  MarshalException                        - error during marshalling operation
  PropertyException                       - error getting or setting a property
  UnmarshalException                      - error during un-marshalling
  ValidationException                     - validation failed
JMException                           - non-run-time exceptions thrown by JMX
                                                                  implementations
  MBeanException                          - user-defined exceptions thrown by MBean agent
    MBeanRegistrationException                - preRegister, preDeregister error detected
  OpenDataException                       - one or more data validity contraints not met
  OperationsException                     - exceptions encountered when performing
                                                                  operations on MBeans
    AttributeNotFoundException                - attribute does not exist or can't be retrieved
    InstanceAlreadyExistsException            - MBean is already registered
    InstanceNotFoundException                 - specified MBean not found in the registry
    IntrospectionException                    - exception occurred during introspection
    InvalidAttributeValueException            - value not valid for attribute
    ListenerNotFoundException                 - specified MBean listener not registered
    MalformedObjectNameException              - format of string does not correspond to a
                                                                  valid ObjectName
    NotCompliantMBeanException                - not a JMX compliant MBean
    ServiceNotFoundException                  - requested service not supported
  ReflectionException                     - java.lang.reflect class methods applied to
                                                                  MBeans not working
  RelationException                       - superclass of all exceptions that can be raised
                                                                  during relation management
    InvalidRelationIdException                - relation id already in use
    InvalidRelationServiceException           - self explanatory
    InvalidRelationTypeException              - duplicate or invalid relation type
    InvalidRoleInfoException                  - min. degree > max. degree
    InvalidRoleValueException                 - number of referenced MBeans outside of
                                                                  range for role
    RelationNotFoundException                 - no relation corresponding to requested
                                                                  relation id
    RelationServiceNotRegisteredException     - self explanatory
    RelationTypeNotFoundException             - self explanatory
    RoleInfoNotFoundException                 - self explanatory
    RoleNotFoundException                     - does not exist, not readable, or not settable
KeySelectorException                  - XML crypto key in error
LastOwnerException                    - attempt to delete the last owner of an ACL
```

# Checked Exceptions, continued

- **LineUnavailableException** - sound line not avaialbe
- **MarshalException** - XML marshalling or un-marshalling error
- **MidiUnavailableException** - MIDI component unavailable or not creatable
- **MimeTypeParseException** - superclass for MIME parsing exceptions
- **NamingException** - superclass for exceptions in Context and DirContext interfaces
  - **AttributeInUseException** - attempt to add an attribute already in use
  - **AttributeModificationException** - attempt to modify an attribure that conflicts with existing state
  - **CannotProceedException** - cannot resolve composite name
  - **CommunicationException** - client unable to communicate with directory or naming service
  - **ConfigurationException** - configuration error of some kind
  - **ContextNotEmptyException** - attempt to destroy a non-empty context
  - **InsufficientResourcesException** - resource shortage on client or server
  - **InterruptedNamingException** - naming operation interrupted
  - **InvalidAttributeIdentifierException** - directory specific error related to attr. ID
  - **InvalidAttributesExceptio** - attempt to add or modify an attribute set that is not valid
  - **InvalidAttributeValueException** - value conflicts with its specified schema
  - **InvalidNameException** - name does not conform to host naming system
  - **InvalidSearchControlsException** - specification of SearchControls for search operation is invalid
  - **InvalidSearchFilterException** - self explanatory
  - **LimitExceededException** - user or system specified limit exceeded
    - **SizeLimitExceededException** - exceeded user / system size/count limit
    - **TimeLimitExceededException** - exceeded user or system time limit
  - **LinkException** - problem encountered resolving link
    - **LinkLoopException** - exceeded limit count or encountered loop in resolving link
    - **MalformedLinkException** - self explanatory
  - **NameAlreadyBoundException** - name cannot be bound to new object
  - **NameNotFoundException** - some name component cannot be resolved because it is not bound
  - **NamingSecurityException** - supperclass of security-related exceptions for naming
    - **AuthenticationException** - authentication failed
    - **AuthenticationNotSupportedException** - authentication type / strength not supported
    - **NoPermissionException** - requester not permitted to perform requested operation

# Checked Exceptions, continued

| | |
|---|---|
| NoInitialContextException | - no initial context can be created |
| NoSuchAttributeException | - attempt to access non-existant attribute |
| NotContextException | - need context to resolve object so far, but it is not a context |
| OperationNotSupportedException | - context implementation does not support the currently requested operation |
| PartialResultException | - request cannot be fully completed |
| ReferralException | - abstract class for referral exceptions |
| LdapReferralException | - self explanatory |
| SchemaViolationException | - request causes violation of schema rules |
| ServiceUnavailableException | - attempt to communicate with service or directory that is not available |
| NoninvertibleTransformException | - request for AffineTransform on object in state where this is not allowed |
| NoSuchFieldException | - class does not contain a field by this name |
| NoSuchMethodException | - class does not contain a method by this name |
| NotBoundException | - look up or unbind name that has no binding |
| NotOwnerException | - when only owner is allowed to perform op. and owner is not requestor |
| ParseException | - unexpected error while parsing |
| ParserConfigurationException | - serious configuration error |
| PrinterException | - exception in the print system |
| PrinterAbortException | - print job aborted without completing |
| PrinterIOException | - I/O error during printing |
| PrintException | - provide string describing error in print system |
| PrivilegedActionException | - action performed threw a checked exception |
| PropertyVetoException | - attempt to change a value of a property to a non-allowed value |
| RefreshFailedException | - refresh operation for credentials failed |
| RemarshalException | - CORBA error |
| SAXException | - general SAX error warning [public domain] |
| SAXNotRecognizedException | - XML reader found unrecognized identifier |
| SAXNotSupportedException | - XML reader unable to perform requested op. |
| SAXParseException | - XML SAX parser error |
| ScriptException | - wrapper for scripting API exceptions |
| ServerNotActiveException | - invocation of getClientHost outside of remote method call environment |
| SOAPException | - general SOAP exception wrapper |

# Checked Exceptions, continued

| | |
|---|---|
| SQLException | - general SQL exception class, for... |
| BatchUpdateException | - error encountered in processing a series of SQL commands |
| RowSetWarning | - warnings from any RowSet implementation |
| SerialException | - error with serialization or de-serialization of SQL types |
| SQLClientInfoException | - one or more client info properties could not be set on a connection |
| SQLNonTransientException | - outstanding error needs to be cleared |
| SQLDataException | - SQL state value 22 returned |
| SQLFeatureNotSupportedException | - SQL state value of x'0A' returned |
| SQLIntegrityConstraintViolationException | - SQL state value of 23 returned |
| SQLInvalidAuthorizationSpecException | - SQL state value of 28 returned |
| SQLNonTransientConnectionException | - SQL state value of 08 returned |
| SQLSyntaxErrorException | - SQL state value of 42 returned |
| SQLRecoverableException | - might be able to fix up and retry |
| SQLTransientException | - might be able to succeed on retry |
| SQLTimeoutException | - self explanatory |
| SQLTransactionRollbackException | - SQL state 40; automatic rollback done |
| SQLTransientConnectionException | - SQL state 08; retry might succeed |
| SQLWarning | - general warning; silently chained to object |
| DataTruncation | - unexpected truncation on write or read |
| SyncFactoryException | - an error in the SyncFactory mechanism |
| SyncProviderException | - an error in the SyncProvider mechanism |
| TimeoutException | - blocking operation has timed out |
| TooManyListenersException | - only one listener allowed on specified event |
| TransformerException | - error during transformation |
| TransformerConfigurationException | - serious configuration error |
| TransformException | - error in transformation algorithm |
| UnmodifiableClassException | - one of specified classes may not be modified |
| UnsupportedAudioFileException | - audio processing error |
| UnsupportedCallbackException | - handler does recognize requested Callback |
| UnsupportedFlavorException | - requested data not supported in this flavor (AWT) |
| UnsupportedLookAndFeelException | - look and feel classes not present (swing) |
| URIReferenceException | - error dereferencing a URI |
| URISyntaxException | - string could not be parsed as a URI |

# Checked Exceptions, continued

| | |
|---|---|
| **UserException** | **- more CORBA-IDL exceptions:** |
| **AdapterAlreadyExists** | **- generated by IDL-to-Java compiler** |
| **AdapterInactive** | **- generated by IDL-to-Java compiler** |
| **AdapterNonExistent** | **- generated by IDL-to-Java compiler** |
| **AlreadyBound** | **- generated by IDL-to-Java compiler** |
| **BadKind** | **- inappropriate op. on a TypeCode object** |
| **Bounds** | **- parameter not in legal bounds** |
| **Bounds** | **- exceed bounds on index exceeding number of members in type** |
| **CannotProceed** | **- generated by IDL-to-Java compiler** |
| **DuplicateName** | **- generated by IDL-to-Java compiler** |
| **FormatMismatch** | **- generated by IDL-to-Java compiler** |
| **ForwardRequest** | **- generated by IDL-to-Java compiler (portable server)** |
| **ForwardRequest** | **- generated by IDL-to-Java compiler (portable interceptor)** |
| **InconsistentTypeCode** | **- generated by IDL-to-Java compiler** |
| **InconsistentTypeCode** | **- attempt to create dynamic array with type code not subclass of DynAny** |
| **Invalid** | **- failure in dyamic any operations** |
| **InvalidAddress** | **- generated by IDL-to-Java compiler** |
| **InvalidName** | **- no initial reference passed** |
| **InvalidName** | **- generated by IDL-to-Java compiler** |
| **InvalidName** | **- generated by IDL-to-Java compiler** |
| **InvalidPolicy** | **- generated by IDL-to-Java compiler** |
| **InvalidSeq** | **- invalid sequence on a Dyn any operation** |
| **InvalidSlot** | **- generated by IDL-to-Java compiler** |
| **InvalidTypeForEncoding** | **- generated by IDL-to-Java compiler** |
| **InvalidValue** | **- not described in doc** |
| **InvalidValue** | **- generated by IDL-to-Java compiler** |
| **NoContext** | **- generated by IDL-to-Java compiler** |
| **NoServant** | **- generated by IDL-to-Java compiler** |
| **NotEmpty** | **- generated by IDL-to-Java compiler** |
| **NotFound** | **- generated by IDL-to-Java compiler** |
| **ObjectAlreadyActive** | **- generated by IDL-to-Java compiler** |
| **ObjectNotActive** | **- generated by IDL-to-Java compiler** |
| **PolicyError** | **- generated by IDL-to-Java compiler** |
| **ServantAlreadyActive** | **- generated by IDL-to-Java compiler** |
| **ServantNotActive** | **- generated by IDL-to-Java compiler** |
| **TypeMismatch** | **- dynamic accessor invalid type** |
| **TypeMismatch** | **- generated by IDL-to-Java compiler** |
| **TypeMismatch** | **- generated by IDL-to-Java compiler** |
| **UnknownEncoding** | **- generated by IDL-to-Java compiler** |

# Checked Exceptions, continued

**UnknownUserException** - client uses DII to make request, error encountered
**WrongAdapter** - generated by IDL-to-Java compiler
**WrongPolicy** - generated by IDL-to-Java compiler
**WrongTransaction** - transaction scope different than original
**XAException** - Transaction manager exception
**XMLParseException** - XML parse error in ModelIMBean context
**XMLSignatureException** - XML signature generation or validation error
**XMLStreamException** - general XML error class
**XPathException** - generic XPath exception class
**XPathExpressionException** - self explanatory
**XPathFunctionException** - self explanatory
**XPathFactoryConfigurationException** - self explanatory

❒ **Remember, the thing about checked exceptions: if you invoke a method that might throw one of these, your code needs to include exception handling logic (**try **block or** throws **clause)**

# Unchecked Exceptions

- **Unchecked exceptions are Error classes and their subclasses, and RunTimeExceptions and their subclasses**

- **Error classes and subclasses hierarchy:**

**Object**

**Throwable**

**Error ...**

| | |
|---|---|
| **AnnotationFormatError** | **- badly formed annotation** |
| **AssertionError** | **- an assertion has failed** |
| **AWTError** | **- Abstract Window Toolkit failure** |
| **CoderMalfunctionError** | **- CharsetEncoder failure** |
| **FactoryConfigurationError** | **- problem with xml Parser Factories** |
| **FactoryConfigurationError** | **- problem with xml stream config.** |
| **IOError** | **- serious I/O Error** |
| **LinkageError** | **- class incompatibility error** |
| **ClassCircularityError** | **- circularity found while initializing a class** |
| **ClassFormatError** | **- invalid class file found** |
| **GenericSignatureFormatError** | **- syntacticlly malformed signature reflective method** |
| **UnsupportedClassVersionError** | **- major+minor version no. not supported** |
| **ExceptionInInitializerError** | **- static initializer error** |
| **IncompatibleClassChangeError** | **- change causes dependency failure** |
| **NoClassDefFoundError** | **- class definition now missing** |
| **UnsatisfiedLinkError** | **- problem with 'native' method** |
| **VerifyError** | **- internal inconsistency or security error** |
| **ServiceConfigurationError** | **- error while loading a service provider** |
| **ThreadDeath** | **- 'stop' issued with zero arguments** |
| **TransformerFactoryConfigurationError** | **- error with transformation factory** |
| **VirtualMachineError** | **- JVM broken or out of resources** |
| **InternalError** | **- unexpected internal error** |
| **OutOfMemoryError** | **- out of storage** |
| **StackOverflowError** | **- application recurses too deeply** |
| **UnknownError** | **- duh!** |

# Unchecked Exceptions, continued

## ❒ RuntimeException classes and subclasses hierarchy:

**Exception**

**RuntimeException ...**

**AnnotationTypeMismatchException - attempt to access annotation whose type has changed**
**ArithmeticException - divide by zero and other arithmetic errors**
**ArrayStoreException - try to store wrong kind of object into an array**
**BufferOverflowException - relative 'put' has reached buffer boundary**
**BufferUnderflowException - relative 'get' has reached source buffer limit**
**CannotRedoException - an UndoableEdit has been told to redo**
**CannotUndoException - an UndoableEdit has been told to undo**
**ClassCastException - attempt to cast an object to a subclass of which it is not an instance**
**CMMException - if native CMM returns an error (Swing)**
**ConcurrentModificationException - illegal concurrent modification detected**
**DOMException - Document Object Model found serious error**
**EmptyStackException - self explanatory**
**EnumConstantNotPresentException - self explanatory**
**EventException - dependent on event method**
**IllegalArgumentException - method has been passed an illegal or inappropriate argument**
**IllegalCharsetNameException - string used as charset name is not valid**
**IllegalFormatException - format string contains illegal syntax**
**DuplicateFormatFlagsException - self explanatory**
**FormatFlagsConversionMismatchException - conversion and flag are contradictory**
**IllegalFormatCodePointException - self explanatory**
**IllegalFormatConversionException - argument type mismatch frmt specifier**
**IllegalFormatFlagsException - illegal combination of flags**
**IllegalFormatPrecisionException - unsupported precision specification**
**IllegalFormatWidthException - unsupported width value**
**MissingFormatArgumentException - self explanatory**
**MissingFormatWidthException - self explanatory**
**UnknownFormatConversionException - self explanatory**
**UnknownFormatFlagsException - self explanatory**
**IllegalSelectorException - channel / provider mismatch**
**IllegalThreadStateException - self explanatory**
**InvalidKeyException - composite name or index into table invalid**

# Unchecked Exceptions, continued

| | |
|---|---|
| InvalidOpenTypeException | - self explanatory (beans context) |
| InvalidParameterException | - self explanatory (security package) |
| KeyAlreadyExistsException | - index of row to be added already exists |
| NumberFormatException | - when converting between numeric types |
| PatternSyntaxException | - syntax error in a regular expression |
| UnresolvedAddressException | - network operation on unresolved socket address |
| UnsupportedAddressTypeException | - socket address error |
| UnsupportedCharsetException | - self explanatory |
| IllegalMonitorStateException | - thread tried to wait on monitor it does not own |
| IllegalPathStateException | - operation performe on path not in correct state |
| IllegalStateException | - method invoked at illegal or inappropriate time |
| AlreadyConnectedException | - attempt to connect to socket channel already connected to |
| CancellationException | - value of a task cannot be retrieved because the task has been cancelled |
| CancelledKeyException | - attempt to use an invalid selection key |
| ClosedSelectorException | - I/O attempt against closed selector |
| ConnectionPendingException | - attempt to connect when connection is already in progress |
| FormatterClosedException | - attempt to use closed formatter |
| IllegalBlockingModeException | - blocking mode operation attempted in incorrect blocking mode |
| IllegalComponentStateException | - AWT component not in legal state for requested operation |
| InvalidDnDOperationException | - (AWT) underlying DnD system not in appropriate state |
| InvalidMarkException | - attempt to reset buffer when its mark is not defined |
| NoConnectionPendingException | - finishConnect socket method invoked before connect method |
| NonReadableChannelException | - try to read from channel not opened 'read' |
| NonWritableChannelException | - try to write to channel not opened 'write' |
| NotYetBoundException | - attempt socket I/O before socket bound to application |
| NotYetConnectedException | - attempt socket I/O before connection estab. |
| OverlappingFileLockException | - self explanatory |
| ImagingOpException | - image filter method cannot process image |
| IncompleteAnnotationException | - attempt to access a new element of an annotation type after type compiled |

# Unchecked Exceptions, continued

| | |
|---|---|
| IndexOutOfBoundsException | - self explanatory |
| ArrayIndexOutOfBoundsException | - self explanatory |
| StringIndexOutOfBoundsException | - thrown by some String methods |
| JMRuntimeException | - superclass for the following: |
| MonitorSettingException | - monitor setting goes invalid while running |
| RuntimeErrorException | - error in agent |
| RuntimeMBeanException | - built by MBeanServer when a call to a MBean method throws an exception |
| RuntimeOperationsException | - agent throws when error encountered performing operations on MBeans |
| LSException | - DOM error |
| MalformedParameterizedTypeException | - reflective method passed bad parameterized type |
| MirroredTypeException | - attempt to access Class for a TypeMirror |
| MirroredTypesException | - attempt to access a sequence of Class objects for TypeMirror |
| MissingResourceException | - self explanatory |
| NegativeArraySizeException | - self explanatory |
| NoSuchElementException | - nextElement method working on Enumeration past end |
| NoSuchMechanismException | - requested XML mechanism not found |
| NullPointerException | - attempt to use null when an object is required |
| ProfileDataException | - error in ICC_Profile object (AWT) |
| ProviderException | - superclass for providers, which subclass more details |
| RasterFormatException | - invalid layout information in Raster (AWT) |
| RejectedExecutionException | - Executor cannot accept requested task |
| SecurityException | - superclass for: |
| AccessControlException | - requested access denied |
| RMISecurityException | - deprecated |
| SystemException | - base class for all CORBA standard exceptions |
| ACTIVITY_COMPLETED | - requested activity completed w/ timeout or error |
| ACTIVITY_REQUIRED | - activity context was not found |
| BAD_CONTEXT | - invalid context for requested activity |
| BAD_INV_ORDER | - invalid sequence of requests |
| BAD_OPERATION | - operation not supported by object |
| BAD_PARAM | - self explanatory |
| BAD_QOS | - object can't support requested quality of service |
| BAD_TYPECODE | - self explanatory |
| CODESET_INCOMPATIBLE | - meaningful conversation not possible between client and server |
| COMM_FAILURE | - communication lost during an operation |
| DATA_CONVERSION | - character set, floating point representation, etc. |
| FREE_MEM | - attempt to free memory failed |

# Unchecked Exceptions, continued

| | |
|---|---|
| IMP_LIMIT | - some implementation limit has been exceeded |
| IndirectionException | - Java-CORBA demarshall failure |
| INITIALIZE | - ORB has failed to initialize properly |
| INTERNAL | - internal ORB failure |
| INTF_REPOS | - failure to reach or read the interface repository |
| INV_FLAG | - invalid flat passed to an operation |
| INV_IDENT | - syntactically invalid IDL identifier |
| INV_OBJREF | - object reference internally malformed |
| INV_POLICY | - incompatiblity between policy overrides |
| INVALID_ACTIVITY | - activity resumed in different context than initial |
| INVALID_TRANSACTION | - request carried an invalid context |
| MARSHAL | - request or reply from network structurally invalid |
| NO_IMPLEMENT | - the operation exists, but is not implemented |
| NO_MEMORY | - ORB run time has run out of memory |
| NO_PERMISSION | - caller has insufficient privileges for request |
| NO_RESOURCES | - some general resource limitation encountered |
| NO_RESPONSE | - response for a deferred synchronous request not available yet |
| OBJ_ADAPTER | - adminstrative mismatch |
| OBJECT_NOT_EXIST | - invocation on a deleted object |
| PERSIST_STORE | - failure of underlying persistent store (database) |
| REBIND | - conflict with RebindPolicy |
| TIMEOUT | - time to live period has been exceeded |
| TRANSACTION_MODE | - ORB client detects mismatch between InvocationPolicy and chosen path |
| TRANSACTION_REQUIRED | - request carried null transaction, but non-null transaction required |
| TRANSACTION_ROLLEDBACK | - request against transaction already rolled back, or marked for rollback |
| TRANSACTION_UNAVAILABLE | - connection has been lost |
| TRANSIENT | - object could not be reached |
| UNKNOWN | - non-CORBA exception thrown |
| UnknownException | - Java-CORBA communication mechanism for unknown exceptions |
| TypeConstraintException | - violation of dynamically checked type constraint |
| TypeNotPresentException | - definition for named type cannot be found |
| UndeclaredThrowableException | - method invocation on Proxy instance throws checked exception |
| UnknownAnnotationValueException | - unknown kind of annotation value encountered |

# Unchecked Exceptions, continued

**UnknownElementException** - could be version level differences
**UnknownTypeException** - could be version level differences
**UnmodifiableSetException** - op. cannot be performed because set is unmodifiable
**UnsupportedOperationException** - requested operation not supported
  **HeadlessException** - environment does not support keyboard nor mouse, but app requires them (AWT)
  **ReadOnlyBufferException** - change attempted against content of read only buffer
**WebServiceException** - base for all JAX-WS API runtime exceptions

# Appendix C - Unicode Work

❒

- ◆

# Special characters

## A



## B

## C

## D



## E

# F

# G



# H



# I

# J



# K



# L

## M



## N



## O

## P



## R

## S



## T

## U



## V



## W

# Z