Objectives 3

API Contents

Using Wrappers (Objective 3.1)

- The wrapper classes correlate to the primitive types.
- Wrappers have two main functions:
  - To wrap primitives so that they can be handled like objects
  - To provide utility methods for primitives (usually conversions)
- The three most important method families are
  - xxxValue() Takes no arguments, returns a primitive
  - parseXxx() Takes a String, returns a primitive, throws NFE
  - valueOf() Takes a String, returns a wrapped object, throws NFE
- Wrapper constructors can take a String or a primitive, except for Character, which can only take a char.

Boxing (Objective 3.1)

- As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
- Using == with wrappers is tricky; wrappers with the same small values (typically lower than 127), will be ==, larger values will not be ==.

Using String, StringBuffer, and StringBuilder (Objective 3.1)

- String objects are immutable, and String reference variables are not.
- If you create a new String without assigning it, it will be lost to your program.
- If you redirect a String reference to a new String, the old String can be lost.
- String methods use zero-based indexes, except for the second argument of substring().
- The String class is final—its methods can't be overridden.

Using String, StringBuffer, and StringBuilder (Objective 3.1) [contd.]

- When the JVM finds a String literal, it is added to the String literal pool.
- Strings have a method: length(), arrays have an attribute named length.
- The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- StringBuilder methods should run faster than StringBuffer methods.

Using String, StringBuffer, and StringBuilder (Objective 3.1) [contd.]

- All of the following bullets apply to both StringBuffer and StringBuilder:
  - They are mutable—they can change without creating a new object.
  - StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
  - StringBuffer equals() is not overridden; it doesn't compare values.
- Remember that chained methods are evaluated from left to right.
- String methods to remember: charAt(), concat(), equalsIgnoreCase(), length(), replace(), substring(), toLowerCase(), toString(), toUpperCase(), and trim().
- Stringbuffer methods to remember: append(), delete(), insert(), reverse(), and toString().

File I/O
(Objective 3.2)

- The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, and PrintWriter.
- A new File object doesn't mean there's a new file on your hard drive.
- File objects can represent either a file or a directory.
- The File class lets you manage (add, rename, and delete) files and directories.

File I/O
(Objective 3.2) [contd.]

- The methods createNewFile() and mkDir() add entries to your file system.
- FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
- It's very common to "wrap" a BufferedReader around a FileReader, to get access to higher-level (more convenient) methods.

File I/O (Objective 3.2) [contd.]

- It's very common to "wrap" a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- Java 5 PrintWriters have new append(), format(), and printf() methods.

Serialization (Objective 3.3)

- The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
- A class must implement the Serializable interface before its objects can be serialized.
- The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.
- If you mark an instance variable transient, it will not be serialized even thought the rest of the object's state will be.

Serialization (Objective 3.3) [contd.]

- You can supplement a class's automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding calls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.
- If a superclass implements Serializable, then its subclasses do automatically.
- If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will run.
- DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

Dates, Numbers, and Currency (Objective 3.4)

- The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
- Most of the Date class's methods have been deprecated.
- A Date is stored as a long, the number of milliseconds since January 1, 1970.
- Date objects are go-betweens the Calendar and Locale classes.
- The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.

Dates, Numbers, and Currency (Objective 3.4) [contd.]

- Create Calendar instances using static factory methods (getInstance()).
- The Calendar methods you should understand are add(), which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and roll(), which works like add() but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
- DateFormat instances are created using static factory methods (getInstance() and getDateInstance()).
- There are several format "styles" available in the DateFormat class.

Dates, Numbers, and Currency (Objective 3.4) [contd.]

- DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- The DateFormat.format() method is used to create Strings containing properly formatted dates.
- The Locale class is used in conjunction with DateFormat and NumberFormat.
- Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
- For the exam you should understand creating Locales using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting (Objective 3.5)

- regex is short for regular expressions, which are the patterns used to search for data within large data sources.
- regex is a sub-language that exists in Java and other languages (such as Perl).
- regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".
- Study the \d, \s, \w, and . metacharacters
- regex provides for *quantifiers* which allow you to specify concepts like: "look for one or more digits in a row."
- Study the ?, *, and + greedy quantifiers.

Parsing, Tokenizing, and Formatting (Objective 3.5) [contd.]

- Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance String s = "\\d";
- The Pattern and Matcher classes have Java's most powerful regex capabilities.
- You should understand the Pattern compile() method and the Matcher matcher(), pattern(), find(), start(), and group() methods.
- You WON'T need to understand Matcher's replacement-oriented methods.

Parsing, Tokenizing, and Formatting (Objective 3.5) [contd.]

- You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
- Tokenizing is the process of splitting delimited data into small pieces.
- In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- Tokenizing can be done with the Scanner class, or with String.split().
- Delimiters are single characters like commas, or complex regex expressions.

Parsing, Tokenizing, and Formatting (Objective 3.5) [contd.]

- The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- The Scanner class allows you to tokenize Strings or streams or files.
- The String.split() method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- New to Java 5 are two methods used to format data for output. These methods are format() and printf(). These methods are found in the PrintStream class, an instance of which is the out in System.out.

Parsing, Tokenizing, and Formatting (Objective 3.5) [contd.]

- The format() and printf() methods have identical functionality.
- Formatting data with printf() (or format()) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- The format() method allows you to mix literals in with your format strings.
- The format string values you should know are

  Flags: -, +, 0, "," , and (
- Conversions: b, c, d, f, and s
- If your conversion character doesn't match your argument type, an exception will be thrown.