WA2892 Booz Allen Hamilton Tech Excellence Modern Software Development Program - Phase 1

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Web Age Solutions Inc.
821A Bloor Street West
Toronto
Ontario, M6G 1M1

Table of Contents

Chapter 1 - Overview of Java

Objectives

After completing this unit, you will be able to:

- Describe the main components of the Java platform
- Identify benefits of the Java platform
- Describe various editions and uses of Java

1.1 History Of Java

- 1992-93 - "Oak" programming language created for interactive TV
- 1995 - Java debuts at SunWorld '95 conference
- 1996 - First release of Java and support in major internet browsers
- 1998 - Release of Java 1.2 (Java 2) with major graphics improvements
- 2004 - Version 5 of Java released
  - Format of version numbers changed
- 2006 - Version 6 of Java released
  - Not many big differences from Java 5
- 2010 – Oracle acquires Sun and takes control of Java
- 2011 – Version 7 of Java released
  - Includes some useful syntax changes for specific situations
  - Not many fundamental changes
- 2014 – Version 8 of Java released
  - Inclusion of long-awaited "lambda expressions"
  - Other more major language changes (similar in scope to Java 5)

History Of Java

Java was not originally designed for the Internet as many believe. The predecessor of Java, a language called Oak, was being developed for use in various consumer electronics devices where a programming language independent of the hardware platform would be necessary. Just as the language was

becoming mature (and as those at Sun realized the demand for programmable consumer electronic devices wasn't as they hoped) the popularity of the Internet started booming. Because the Internet was also a medium which would benefit from a platform independent language, Oak was renamed Java to avoid conflict with another product named Oak, was adapted to deliver interactive programs over the internet (Applets), supported by nearly ever major Internet browser, and Java was born! The Java language just happened to be in the right place at the right time.

Starting with version 5 the format of the version numbers was changed so that the number before the decimal changed with each version. Java went from Java 1.4 to Java 5 but this was just one release different.

1.2 Benefits Of Java

- Platform Independence
  - The Java platform is available for Windows, Mac, Unix, Linux, PDAs, and even cell phones!
  - Provides a standard environment to run programs
- Portability
  - Programs written and compiled on one hardware platform may be run on another without being modified
  - "Write once, run anywhere!"
- Object Oriented
  - Solve problems in terms of the objects and entities involved
- Open
  - Although maintained by Oracle, Java is part of a community process where members provide input and improvements

Benefits Of Java

There are several other buzzwords that can be associated with Java:

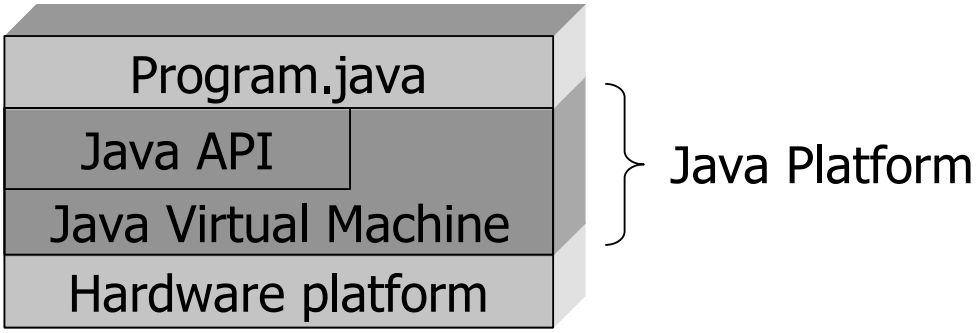
- Simple
- Object Oriented
- Distributed
- Interpreted

- Robust
- Secure
- Architecture Neutral
- Portable
- High Performance
- Multithreaded
- Dynamic

For a more complete discussion of some of these, check out "The Java Language Environment" whitepaper at: http://java.sun.com/docs/white/langenv/

1.3 What Is Java?

- The "Java Platform" has several components
  - The Java programming language
  - The Java "Virtual Machine"
  - The Java Application Programming Interface (Java API)
- The term "platform" refers to the Java software environment



What Is Java?

Many times the Java platform is described as only the Java Virtual Machine and the Java API, as shown in the figure above. In the slide the Java programming language is included because you must write your programs in the Java language. Because Java needs your program to actually do anything, the language is a fundamental part of the Java platform.

The Java API is a rich set of software libraries for doing everything from input and output, networking, and security to relational database access and complex graphics.

1.4 What's This "Virtual Machine"?

- Java is first compiled into machine independent "bytecode"
- This bytecode is interpreted by the virtual machine (or VM) when the program is run
- The VM performs all the tasks of a real processor in a safe, virtual environment
- The VM transforms the instructions into versions for the specific hardware

What's This "Virtual Machine"?

Unlike some other languages, the specification for the Java language and the Java virtual machine are complete and open specifications. This allows anyone to create a Java virtual machine for a specific hardware environment. This could be as simple as providing Java software support for a new operating system or as complex as creating devices (like watches, smart-cards, or wearable devices) with Java support built directly into the hardware.

Also unlike other programming languages, Java is both compiled and interpreted. This allows for error-checking during compilation into an executable program but also allows the flexibility of being interpreted when executed. The latest improvements to Java help ensure that the runtime interpretation of code does not cause performance to suffer greatly. This improvement, called just-in-time compilation (or JIT), allows a runtime system to optimize performance by compiling byte code to native machine code on the fly.

1.5 Comparison to Other Languages

- Strict type safety
  - You must declare the type of a variable before using it ensuring the rest of the program always uses the variable correctly
- Automatic memory management
  - Java can automatically remove objects in memory that are no longer needed through a process of "garbage collection"
- No direct access to OS-level functions

- All access to OS-level functionality is done through the "Java Virtual Machine"
- You can still do things like work with files, network connections, etc but the JVM translates instructions for the OS

Comparison to Other Languages

Although it would be impossible to compare Java to every other language the above features generally distinguish Java from several other common programming languages.

1.6 Java Programs

- Applications
  - Normal, standalone programs
    - Console applications – text based input and output
    - Windowed applications – Graphical User Interface (GUI) mechanisms such as menus and toolbars
- Applets
  - Interactive programs embedded in web pages
  - Executed using a web browser or the Java Applet Viewer
- "Server-side" applications
  - Clients interact with applications that execute on another machine, the "server"

1.7 Basic Java Development Tools

- Compiler
  - 'javac' command
  - Compiles Java source code into executable bytecode
- Application launcher
  - 'java' command
  - Executes Java applications
- Applet viewer
  - 'appletviewer' command
  - Runs Java Applets without a web browser
- API documentation generator
  - 'javadoc' command
  - Creates HTML pages documenting classes
  - Parses information from special comments within source code
- Debugger
  - 'jdb' command
  - Executes code in a special VM which allows you to step through code

Basic Java Development Tools

All of these development tools (and others) are part of the freely available software development kit (or SDK) for Java. Prior to version 1.2, this kit was termed the Java Development Kit (or JDK). You may still see the term JDK used to refer to the set of basic development tools.

1.8 Java Editions

- Java SE – "Java Standard Edition" – Set of standard libraries for Applications and Applets
  - This is the primary edition we will learn
  - Most recent: Java SE 8
- Java EE – "Java Enterprise Edition" – Set of extensions for running applications on web servers
  - Most recent: Java EE 7
- Java ME – "Java Micro Edition" – Set of libraries for portable devices designed for the limited resources of such devices
  - Multiple technologies: most v1.1 or higher

1.9 Example – HelloWorld

- The code below is a complete (albeit simple) Java program:

```
public class HelloWorld {
	public static void main(String[] args)
	{
		System.out.println("Hello World!");
	}
}
```

1.10 Java Classes

- Java organizes code into "Classes"
- A "Class declaration" defines the name
- The following code defines the 'HelloWorld' class

```
public class HelloWorld {
	...
}
```

- Everything within the curly brackets { } is part of the class definition

1.11 Main Methods

- A "main" method defines how a Java program executes
- This method always has the same "signature" shown below

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

- Everything within the curly brackets executes when the program executes

1.12 Statements

- Java statements provide specific instructions
- A Java program can be viewed as a sequential execution of statements

- Statements end with a semi-colon ';'
- The following statement tells Java to print "Hello World!" to the system output:

```
System.out.println("Hello World!");
```

1.13 Summary

- Most of Java's appeal comes from being independent of specific hardware platforms
- The Java platform is comprised of the Java language, Java virtual machine and Java APIs
- Once compiled into executable "bytecode" Java programs may be run on other systems without modification
- Applications and applets are two types of Java programs
- There are several "editions" of Java which are intended for various uses

Chapter 2 - Basic Object Concepts

Objectives

After completing this unit, you will be able to:

- Describe some of the basic concepts of object-oriented programming
- Understand the difference between a class definition and an object
- List some of the benefits of using an object-oriented programming language

Objectives

The main purpose of this section is to introduce some of the terminology involved along with broad definitions of the terms. Further details about all of these concepts will be provided in other sections as the course progresses.

2.1 What Is An Object?

- Anything can be thought of as an object
- Software abstraction which bundles together data (state) and function (behavior)

- Video tapes in a video store
- A bank account
- A dog
- A book

What Is An Object?

Several slides will use a similar layout where a definition is given on the left (with blue square bullets) and examples are given on the right (with red diamond bullets).

2.2 State

- Set of attributes that describe the condition of an object
- State of two objects may be identical but remain separate objects
- Video tape: movie on tape, running time, cast, rating
- Bank account: balance, account type, transactions, account number
- Dog: breed, weight, age, color, height, sex

2.3 Behavior

- Set of operations that either access or modify the state of an object
- An action taken by an object in response to a message or a state change
- Video: rewind video
- Bank account: make deposit
- Dog: bark
- Book: turn page

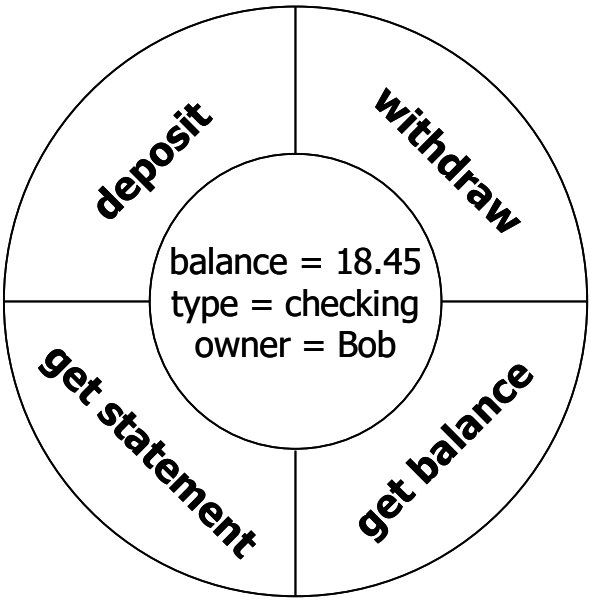
Behavior

Typically it is another object that executes a function of an object. For example, a VCR would rewind a video tape, a person would deposit money into a bank account or turn the page of a book. On the other hand, the dog may instruct itself to bark based on the proximity of other dogs or noises it hears.

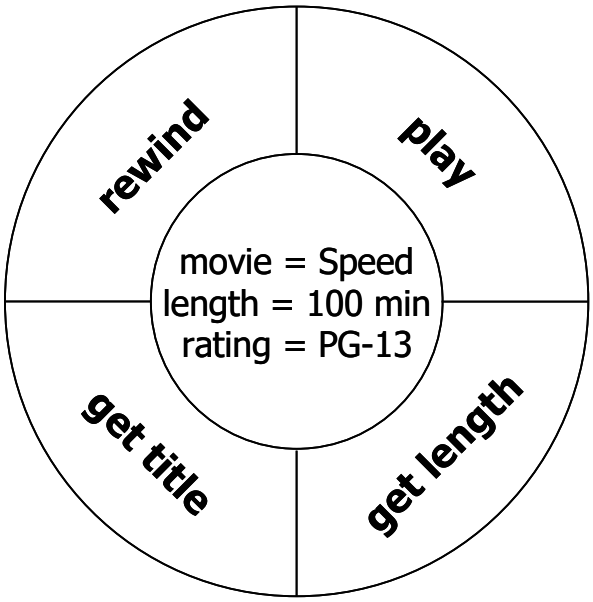
2.4 Encapsulation

- The mechanism that bundles together state and behavior
- Self-contained definition which hides the details from outside entities
- Definitions of "complete" logical objects
- We do not need to know exactly how a video tape works, just how to use it
- We do not need to know the procedure a bank follows when making a deposit, just what the account balance is

2.5 Encapsulation Examples



A Bank account object



A Video tape object

2.6 Classes vs. Objects

- Class
  - Abstraction describing common features of all members of a group
  - Template for creating, or "instantiating", object instances
  - A VideoTape class might define overall characteristics of tapes
- Object
  - Individual "instances" with states and identities
  - A video store has many tapes which are all similar but may be rented separately and contain different movies

Classes vs. Objects

Although only objects are created when a program runs, every object is an "instance" of a class. This means that every object will have a corresponding class definition which defines the type of state and behavior encapsulated within an object. These class definitions are what is created when designing a Java system. The component nature of classes allows us to define several small pieces of the system

and then fit them all together. This also aids parallel development in a team environment.

The "VideoTape" class is identified using one word because class definitions must be named with one word identifiers. The details of this will be covered later in the course.

2.7 Inheritance

- Define a new class based on the differences from another class
- New class inherits all attributes and behaviors from the "parent" class
- Define superclass of RentalItem, and subclasses Video, DVD, and Game
- RentalItem contains the definition common to all items for rent while the subclasses contain the differences

Inheritance

A "subclass" inherits from a "superclass" and is lower in the hierarchy of class definitions.

Another example of inheritance might be found in the bank accounts example. There may be a common superclass called BankAccount which defines common features like account number, account owner, and owner address. Inheriting from this superclass might be subclasses like a CheckingAccount with a debit card, a SavingsAccount with an interest rate, and a LineOfCredit with a maximum balance.

Inheritance makes for more manageable code because making changes to the RentalItem or BankAccount definition only requires change in one place, not in each subclass that inherits from these superclasses.

2.8 Interfaces

- Abstract definition of common features
- Interfaces do not contain state, only behavior
- Instances of the interface do not exist, it is simply a way to define commonality
- To utilize an interface, it must be "implemented" by another class which includes the specifics on the behaviors defined in the interface
- An Animal might share common behaviors but may not share common state
- RentalItem is used to define the common functions (like rent or return) but everything in the store is a Video, DVD, or Game
- There may be some state that all items for rent would share which might make a RentalItem interface inappropriate

Interfaces

Although you may not have an object instance of an interface type, you may declare a variable to be of an interface type. This means that if RentalItem is declared as an interface, you may have a variable declared as a RentalItem type (say to rent or return the item) but the actual object will remain a Video, DVD, or Game. This allows you to create a program to deal with the more generic types instead of adding code every time a new type of item to rent is made available.

2.9 Polymorphism

- Ability of two or more objects of different classes to respond to the same message (method) in different ways
- Single name may express different behavior
- "Multiple personality disorder"
- Customer has list of RentalItems to be checked in but the process for check-in is different depending on the type of Rental Item (Video, DVD, etc)
- Telling a Dog to bark would give a different result for a Pug than a GermanSheppard

Polymorphism

The concepts of inheritance, interfaces, and polymorphism are closely linked. Superclasses and

interfaces allow for you to use more generic types in a program but still have specific behavior for each individual subclass.

2.10 Benefits Of Objects

- Natural – program in terms of the elements involved in a problem
- Reliable – modular nature increases reliability
- Reusable – definitions from one Program may be reused to solve a similar problem
- Maintainable – errors reduced to one area that may be easily fixed without modifying other definitions
- Extendable – easy to add functionality at later dates
- Timely – increases parallel development which reduces overall time of development

2.11 Summary

- An object is a natural way to think of elements of a problem
- Objects have both state and behavior
- A class completely defines the state and behavior exhibited by a group of objects
- Several class definitions may inherit state and behavior from one common "superclass" definition
- Several objects may respond to the same message in different ways depending on their type

Chapter 3 - Basic Java Syntax

Objectives

After completing this unit, you will be able to:

- Declare Java variables
- Describe Java primitive types
- Write readable Java code with whitespace and comments
- Understand how statement blocks affect the scope of a variable

3.1 Declaring And Initializing Variables

- Variable names, or identifiers, begin with a letter, underscore (_), or a dollar sign ($)
- The digits 0-9 may be used except to begin an identifier name
- Keywords and reserved words cannot be used
- Identifiers are case sensitive
- Spaces may not be used in an identifier
- Declare the type, name, and initial value (optional) of a variable
  - Using an explicit value after the '=' sign is a "literal" value

```
double myVariable = -42.76;
int my2ndVariable;
my2ndVariable = 35;
```

Declaring And Initializing Variables

You must declare the type of a variable in Java. This is often referred to as "strongly typed". If you do not declare the type of a variable and simply try to begin using a new variable name the code will not compile.

A variable must always be initialized before being used. The statement above referring to the initial value as “optional” refers to initializing a variable in the same statement that it is declared. You may initialize a variable when it is declared or in a separate statement (as shown in the last two lines of the example above).

Examples:

```
lastName    // legal
_number1    // legal
myClass     // legal: embedded words are OK
$temp       // legal: starts with a dollar sign
interface   // illegal: a keyword
%num        // illegal: must start with a letter, _ or $
month       // legal
MONTH       // legal: not the same as month:
```

3.2 Keywords

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	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

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Keywords

Java reserves some identifiers for use in naming data types and program operations.

You cannot use Java's reserved identifiers - also known as keywords or reserved words -- to name variables, methods, classes, interfaces or packages.

3.3 Coding Tips – Variables

- Descriptive names
  - Names should be short yet descriptive enough to be understood when used later in code
- Start lowercase
- Capitalize successive words
- Use single character names only with “throwaway” variables (loop, array index)
- Avoid underscore (_) or dollar sign ($)
- Initialize variable when declared

Coding Tips – Variables

Some examples of variable names following these conventions are:

int numberOfPlayers;

char userResponse;

double amountPaid;

double balance;

3.4 Primitive Data Types

- The built-in "primitive" data types of Java are the foundation of most data
  - Integral types (no decimal, "whole" numbers)
    - byte
    - short
    - int
    - long
  - Floating point types (decimal numbers)
    - float
    - double

- Character types (individual characters)
  - char
- Logical types
  - boolean

Primitive Data Types

Java uses primitive types to store simple values. Typically these values will be used in calculations later in the program. The type of calculation depends on the type of value as we will see later with operators.

3.5 Logical - boolean

- There are only two literal values:
  - true
  - false
- Only relational and logical operators can manipulate boolean data types
- No casting is allowed between boolean and non-boolean types
- If not initialized, a boolean variable defaults to false

Logical – boolean

```
// declares a variable status as boolean
// and assigns the value of true

boolean status = true;
```

Note: Relational and logical operators will be discussed later.

3.6 Textual - char and String

- char
  - Represents a 16-bit Unicode character

- Must have its literal enclosed in single quotes
  - ‘a’
  - ‘$’
  - ‘\n’
- If not initialized, a char variable defaults to '\u0000'

- String
  - Represents a sequence of characters
  - **Not a primitive data type. It is a class!**
  - Has its literal enclosed in double quotes
    - `“This is a String!”`

Textual - char and String

Some characters require a special representation to be assigned to character type variables (or Strings as we’ll see later). These special characters require an “escape sequence” so that Java is aware you want to assign one of these special characters to a variable and are not used as part of the Java syntax. These special characters and escape sequences are:

\b – Backspace

\f – Form feed

\n – New line

\r – Carriage return

\t – Tab

\\ - Backslash

\’ – Single quote

\” – Double quote

All characters in Java are stored as Unicode. Since Unicode characters can not be entered directly by ASCII text editors, you may represent a Unicode character by preceding the four hexadecimal digits of the character with the escape sequence \u. For example, ‘\u48d2’ would be a Unicode character.

The char literal can be expressed by:

(1) Enclosing the desired character in a single quote, such as:

```
char desiredLetter = 's';
```

(2) Using the four Unicode hexadecimal digits, preceded by \u, with the entire expression in a single quotes.

```
char letterAInUnicode = '\uXXXX';
```

(3) Using the escape sequence for denoting special characters.

```
char newLine = '\n';
```

Some examples of declaration and initialization of the String type:

// declare a String variable and is initialized

```
String company = "Web Age Solutions Inc";
```

// declare two String variables

```
String firstName, lastname;
```

3.7 Integral - byte, short, int, long

- Format and range of each integral types are:
  - byte 8 bits -2^7 to $2^7 - 1$
  - short 16 bits -2^{15} to $2^{15} - 1$
  - int 32 bits -2^{31} to $2^{31} - 1$
  - long 64 bits -2^{63} to $2^{63} - 1$
- Range is platform independent
- Uses three forms - decimal, octal and hexadecimal
- Default of a literal is int
- Define long by using L or l
- If not initialized, an integral variable defaults to 0 or 0L
- Literal values:
  - 6 (int)
  - 76L (long)

Integral - byte, short, int, long

The Java VM classifies a byte item as a signed 8-bit integer. In this format, the leftmost bit is reserved as the sign bit (0 for positive and 1 for negative). The remaining 7 bits identify the number's magnitude. If the sign bit is 0, the remaining bits are converted from binary to decimal. If the sign bit is 1, the values of the remaining bits are flipped - 1 is converted to 0 and vice versa - and 1 is added to the result.

The value 28 can be expressed in three ways:

(1) The default is decimal.

28

(2) To indicate octal, prefix the literal with zero.

034

(3) To indicate hexadecimal, prefix the literal with 0x, the hex digits may be upper or lower case.

0x1C

0X1c

The following indicates the value of "100" will be represented as a long:

100L or 100l

3.8 Floating Point - float and double

- Format of Floating Point
  - float 32 bits
  - double 64 bits
- Default of a literal is double
- Floating point literals include a decimal point or either of the following
  - E or e (add exponential value)
  - F or f (float)
  - D or d (double)
- If not initialized, a floating point variable defaults to 0.0f or 0.0d
- Literal values:

- 3.54 (defaults to double, same as 3.54d)
- 6.4E4
- -4.7F (float)

Floating Point - float and double

The Java VM classifies a float data item as a sequence of 32 bits. The leftmost bit is the sign bit (0 for positive and 1 for negative). The next eight digits hold the exponent and the final 23 bits holds the mantissa, resulting in about 7 decimal digits of precision. The range is approximately -3.4×10^{38} to 3.4×10^{38}.

The Java VM classifies a double data item as a sequence of 64 bits. The leftmost bit is the sign bit (0 for positive and 1 for negative). The next eleven digits hold the exponent and the final 52 bits holds the mantissa, resulting in about 17 decimal digits of precision. The range is approximately -3.4×10^{308} to 3.4×10^{308}.

The smallest non-zero value that can be represented is roughly $\pm 4.9 \times 10^{-324}$.

The floating point literal can be expressed as:

(1) // declare a float and assign a value of 3.58

```
float num1 = 3.58f;
```

(2) // declare a double and assign the same value of 3.58

```
double num2 = 3.58;
```

3.9 Java 7 – Changes in Numeric Literals

- Sometimes when using numeric literals (hard-coded values) it is difficult to know if the correct number of digits is used
  - eg. having only 15 digits instead of 16 in a credit card number
- Java 7 allows you to place underscore characters (_) between any two digits in a literal value to make it easier to read without changing the value

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
long hexBytes = 0xFF_EC_DE_5E;
```

- You can't place underscores next to a decimal, a literal prefix or suffix, or at the end of a literal

```
float pi2 = 3._1415F; // Invalid;
```

```
long socialSecurityNumber1 = 999_99_9999_L; // Invalid;
int x3 = 52_;  // Invalid;
```

- Java 7 also adds the ability to use binary literals when adding the prefix '0b' or '0B'

```
byte aByte = (byte)0b00100001;
short aShort = (short)0b1010000101000101;
int anInt1 = 0b10100001010001011010000101000101;
// A 'long' value. Note the "L" suffix:
long aLong = 0b0110100001010001011010000101000101L;
```

Java 7 – Changes in Numeric Literals

One of the more common usages of underscores in literals might be between groups of three in place of commas.

Notice that like other numeric literals, the binary literal is interpreted as an 'int' value unless cast to 'byte' or 'short'.

3.10 Strings

- Java class with special support
- Holds combination of 16-bit Unicode characters
- Can be initialized with literals like primitive types
- Several Strings can be combined with “concatenation” (using + operator)

```
String statement = “This is a String!”;
String fullName = "John" + " " + "Smith";
```

Strings

In the second line above the middle element in the statement is just a space character surrounded by double quotes. If this were not added between the other elements they would be combined with no space between.

3.11 Creating Strings

- There are two ways to create objects of the String class

- String literals

```
String statement = “This is a String literal!”;
```

- Using the **new** keyword

```
String secondStatement = new String(“This is a String!”);
```

- Using String literals is the easiest and most common method

Creating Strings

Using the new keyword makes use of a special method of a class called a "constructor". We will discuss constructors in more detail later but for now know that it is a way to create an object, or "instance", of a class with certain initial conditions. In the case of a String, the initial condition is the sequence of characters stored by the String object.

Strings may also be created by a number of other constructors. Most of these alternatives involve arrays of char values or arrays of byte values representing the ASCII codes of the characters to use.

Once created, String objects are "immutable" meaning they can't be altered. The same variable may be reused to refer to another String but this creates a second String object and discards the first.

3.12 White Space

- Java ignores white space and new lines in Java code
- This means we can use this to format programs so they are more readable
- The following is the HelloWorld program but is not as readable

```
public class HelloWorld { public static
void main(String[] args) {
System.out.println("Hello World!"); } }
```

- Format code so it is easily read by others

3.13 Comments

- Comments can be added to code to explain variables, statements, or methods. The three types are:
  - Single-line comment

```
// This is a one line comment
```

- Multi-line comment

```
/* This comment spans multiple lines.
Notice the end. */
```

- Documentation comments
  - Similar to multi-line but start with "/**". Used to create documentation web pages using the "javadoc" utility

Comments

Comments are used to add information to your code to make it more readable to others. The Java compiler will ignore comments in your program and only interpret the program statements. The start of a one line comment (//) tells the compiler to ignore the rest of the current line. The start of a multi-line comment (/*) tells the compiler to ignore everything until reaching the end of the comment (*/) even if that includes several lines.

Single-line comments can be added on the same line as other statements. This is often the easiest way to add comments concerning variables and statements. For example:

```
String firstName;   // used to store the name entered by the user
```

Comments can also be used to temporarily "disable" statements within programs. If the start of a single line comment (//) is added to the beginning of a line, the statement on that line will not affect the program when compiled. This is often helpful to debug programs and track down where errors occur.

3.14 Coding Tips - Comments

- Provide general description of classes and methods
- Describe attributes and variables when identifier names are insufficient
- Describe complex operations
- Outline major steps during methods
  - Blank lines are typically added between these comments and previous statements
- Use blank lines and indentation to separate comments from program code

3.15 Java Statements

- Simple statements
  - End with semi-colons
  - May contain white space and span multiple lines

```
double myVariable = -42.76;
System.out.println(myVariable);
```

- Statement blocks
  - Several statements can be enclosed by curly brackets { } to form a statement block

Java Statements

When enclosing a set of statements with curly brackets to form a statement block, the Java compiler views the block as a single statement in relation to other elements outside of the block. For instance, if you write:

```
if (thisIsTrue) {
          statement 1;
          statement 2;
          statement 3;
}
```

The compiler views it as:

```
if (thisIsTrue)
          Block of statements
```

Although you may space statements over multiple lines, you should avoid doing so unless needed. After all:

```
String name
          = "Susan";
```

is less readable than:

```
String name = "Susan";
```

3.16 Coding Tips - Statements

- One statement per line
- Separate logical groups of statements using blank lines

- Statements within statement blocks should be indented one level from the brackets

```
if (thisIsTrue) {
        statement1;
        statement2;
}
```

Coding Tips - Statements

It is also fairly common to place both brackets of a statement block on the same indentation level as:

```
if (thisIsTrue)
{
        statement1;
        statement2;
}
```

The above example allows you to visually match the sets of brackets and ensure a bracket is not accidentally omitted. Most development tools have utilities to help ensure brackets are matched properly so the first example from the slide is generally preferred to achieve a closer association between the statements of the block and the statement immediately preceding the block.

3.17 Scope of a Variable

```
public class MyClass {
        String firstName;                       (1)
        static int totalCount;                  (2)
        public void myMethod() {
                int b = 4;                      (3)
                {
                        int c;                  (4)
                        c = totalCount + b;     (5)
                }
                totalCount = c;      // error   (6)
        } // end of myMethod
} // end of myClass
```

Scope of a Variable

Based on the above given code:

(1)The variable firstName is defined outside any method. It is accessible within the class. It is called

an instance variable.

(2) The variable totalCount is defined outside any method with the modifier static. It is accessible within the class. It is called a class variable.

(3) The variable b is defined within the method myMethod(). The variable is accessible within myMethod().

(4) The variable c is defined within the inner block of myMethod(). The variable is only accessible within that block, but not in the whole method.

(5) Manipulating totalCount, b, and c is acceptable here.

(6) This will cause an error. The variable c is not accessible outside the block.

3.18 System.out/System.in

- System.out is a basic way to produce output
  - You can use the print or println methods
    - println automatically prints a new line character

```
System.out.println("Hello World!");
```

  - There is also a printf method that can be used for printf style formatted output

```
System.out.printf("Account balance: $ %.2f\n", balance);
```

    - You would need to add a '\n' newline character yourself
- System.in could be used for reading input
  - It is not very easy to use and there are other ways to get input from a user

3.19 Scanner Class

- The java.util.Scanner class can be used as a better way to get basic input
  - This can be created based on System.in

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

  - There are nextXXX() methods to get various primitives

```
int i = input.nextInt();
```

```
String line = input.nextLine();
```

- Make sure to close the Scanner

```
input.close();
```

Scanner Class

More complex ways to get input and produce output will be introduced later.

You can also control the "delimiter pattern" used to separate elements. By default this is "whitespace" which includes spaces, tabs, new lines, and form feeds.

The 'next' methods that return a primitive will not consume the end of a line entered by the keyboard. This means that a call to the 'nextLine' method after one of the other 'nextXXX' methods may pick up the end of the previous line and not block waiting for the user to type another line of text as desired. You can avoid this by making an extra call to 'nextLine' after the other 'nextXXX' methods.

3.20 Summary

- Java variables are always declared with a type
- Some of the most basic types of variables are "primitive types"
- You can add comments, line breaks, and whitespace in your code to make it easier for other programmers to read
- A variable is only available in the "scope" where it was declared

Chapter 4 - Operations and Making Decisions

Objectives

After completing this unit, you will be able to:

- Perform calculations using operators
- Convert primitive values from one type to another
- Determine which operators are evaluated before others
- Make basic decisions based on logical operations

4.1 Operator Categories

- There are several different categories of operators in Java:
- Binary
- Relational
- Bitwise
- Assignment
- Unary
- Logical
- Shift
- Ternary

4.2 Special Situations

- There are also several different situations to consider with operators:
  - Integer Division
  - Numeric Promotion
  - Type Conversion
  - Overflow and Underflow
  - Calculation Errors
  - Operator Precedence

4.3 Binary Operators

- Usual arithmetic operators

- op1 + op2 (addition)
- op1 - op2 (subtraction)
- op1 * op2 (multiplication)
- op1 / op2 (division)
- op1 % op2 (remainder or modulo)

- These are binary operators because they affect two operands

Binary Operators

The above expressions "op1" and "op2" are used to indicate the location of operands involved in the operation. These will be used on several slides describing operators.

The modulo (or remainder) operator is a binary operator that is not as familiar as the other four binary operators. The result of performing this operation is the remainder of diving the first operand by the second operand. Some examples of this are:

11 % 3 = 2

11 % -3 = 2

-11 % 3 = -2

9 % 3 = 0

9 % -3 = 0

Double values may also be used with the modulo operator. For example:

7.5 % 2.0 = 1.5

10.4 % 2.1 = 2.0

-10.4 % 2.1 = -2.0

Notice the answer is always the same sign as the first operator. This is because the result is calculated according to the following formula:

x % y = x - ((int) (x / y) * y)

4.4 Integer Division

- When using the division operator with two variables of an integer type, the remainder is discarded
- This has the effect of rounding towards zero

- For example:
  - 11 / 3 = 3
  - 10 / 6 = 1
  - -16 / 3 = -5
  - 16 / -5 = -3

Integer Division

Integer division behaves the same way for any of the four primitive integer types.

Notice when the result is negative, the result is rounded up to the nearest integer. When the result is positive, the result is rounded down to the nearest integer.

4.5 Numeric Promotion

- When an operation is performed, one or both operands may be "promoted" to a higher type
- Promotion uses the following hierarchy:
  - If either operand is a double, the other is converted to a double
  - If either operand is a float, the other is converted to a float
  - If either operand is a long, the other is converted to a long
  - Otherwise, both are converted to an int
- The result of the operation is the same type as the type of the converted operands

Numeric Promotion

Below are some examples of numeric promotion. This is not a complete list of combinations but any lower type can be substituted for the lower of the two operands without changing the result. The addition operator is simply used as an example, the results would be the same with any binary operator. Notice that for the first example, the type of the result is higher than the type of either operand.

(byte) + (short) = (int)

(short) + (int) = (int)

(int) + (long) = (long)

(long) + (float) = (float)

(float) + (double) = (double)

Variables of types byte and short are not used very often in actual code. Numeric promotion is the primary reason for this as the result of using either of these types in an operation results in a value of type int. Trying to assign this result of an operation to a variable of type byte or short without conversion would result in a compiler error. The minimal gain in saved memory is typically not worth the addition of such computational headaches. Variables of type int are by far the most popular integer type with long types used only for numbers too large for type int.

4.6 Type Conversion Of Primitive Types

- Automatic conversion by numeric promotion
  - Java will automatically convert a value from one type to another as long as there is no possibility of losing information
- Primitive types will be automatically converted according to the following sequence:

```
byte → short → int → long → float → double
```

- Automatic conversion by assignment

```
byte smaller = 5;
int larger = smaller;
```

- Explicit conversion
  - Often called "casting"

```
int larger = 5;
byte smaller = (byte) larger;
```

Type Conversion Of Primitive Types

To convert a value to a smaller type, you must explicitly cast the value to the desired type, as shown in the second example above. If the value is larger than the maximum value for the smaller type, information will be lost without halting execution of the program.

Another use of explicit casting is to force a type conversion to alter the result of an expression. This is most common to avoid losing remainders during integer division. In the following example, both results are double types but the explicit cast affects the final value:

int three = 3;

int two = 2;

double x = 1.5 + three / two // x is assigned a value of 2.5

double y = 1.5 + (double) three / two // y is assigned a value of 3.0

4.7 Unary Operators

- Affect a single operand
- Sign operators (+op, -op)
  - Affect sign of operand
- Increment/Decrement operators (++, --)
  - Increase or decrease the value of an operand by 1
  - Both prefix (++op, --op) and postfix (op++, op--) versions
  - Placement affects sequence of evaluation. Prefix modifies operand value before returning result, postfix modifies operand value after returning result

```
int y;
int x = 10;
y = ++x;      // The value assigned to y is 11
x = 10;
y = x++;      // The value assigned to y is 10
```

Unary Operators

Sign operators are slightly more complex than arithmetic signs. The unary '+' operator will convert values of type byte, short, or char to type int without affecting the sign of the value. The unary '-' operator produces a value with the opposite sign of the operand. If the operand was a negative value, applying the unary '-' operator results in a positive value.

Although the prefix and postfix forms of the increment/decrement operators both produce the same resulting value of the operand, they behave differently when used in expressions. The prefix forms modify the value before returning a value to be used in the expression. The postfix forms modify the value after returning a value to be used in the expression. The following statements demonstrate this behavior:

int y;

int x = 10;

y = ++x; // The value assigned to y is 11

x = 10;

y = x++; // The value assigned to y is 10

x = 10;

y = --x; // The value assigned to y is 9

x = 10;

y = x--; // The value assigned to y is 10

4.8 Relational Operators

- Compares two values and determines the relationship between them

	Use	Returns *true* if
>	*op1* > *op2*	*op1* is greater than *op2*
>=	*op1* >= *op2*	*op1* is greater than or equal to *op2*
<	*op1* < *op2*	*op1* is less than *op2*
<=	*op1* <= *op2*	*op1* is less than or equal to *op2*
==	*op1* == *op2*	*op1* is equal *op2*
!=	*op1* != *op2*	*op1* is not equal *op2*

```
int anInt = 6;
double aDouble = 7.0;
boolean isGreater = anInt > aDouble;
       // isGreater assigned a false value
```

Relational Operators

Relational operators are primarily used in combination with conditional statements or loops. Depending on the result returned by the relational operator, the program may follow one path or another. You may also store the result of the relational operator as the value of a boolean variable and use the result later in the program.

Relational operators follow the same rules of numeric promotion as other operators. For example, the following code would convert both values to double types before making the comparison:

int anInt = 6;

double aDouble = 7.0;

boolean isGreater = anInt > aDouble;

// isGreater assigned a false value

4.9 Logical Operators

- Used to combine several boolean expressions into one expression

	Use	Operator Name
&	*op1* & *op2*	logical AND
&&	*op1* && *op2*	conditional AND
|	*op1* | *op2*	logical OR
||	*op1* || *op2*	conditional OR
!	!*op1*	logical negation (NOT)

- Logical versions always evaluate both operand expressions. Conditional AND (&&) only evaluates the 2nd expression when the 1st is true. Conditional OR (||) only evaluates the 2nd expression when the 1st is false

Logical Operators

Although the logical and conditional versions of the two operators produce the same end result, the difference is in how they are evaluated. The logical version (& and |) will always evaluate both sides of the expression. The conditional versions (&& and ||) will only evaluate the second operand of the expression if the first operand is not enough to determine the result of the operation. For example, if the first operand of a conditional AND operation (&&) is false, the end result will be false no matter what the second operand is so it is not evaluated. For this reason, the conditional versions are sometimes said to "short circuit" evaluation of the expression since they sometimes avoid evaluating the second operand.

Your choice of which version to use will often depend on whether you wish for the second operand to be evaluated every time. The following code would always evaluate the second operand, and therefore always increment the count variable:

```
if (value < 5 & count++ < limit) {
          // Do some work
}
```

The conditional versions can be used to avoid errors that might occur when evaluating the second operand. The following code will avoid errors caused by dividing by zero:

```
if (value != 0 && total/value < 3) {
         // Do some work
}
```

4.10 "Short Circuited" Operators

- The "Conditional" logical operators **&&** and **||** are often called "short circuited"
- They only evaluate the second part of the expression if required
- This can be used to prevent errors

```
int divisor = 0, value = 25;
if (divisor != 0 && value/divisor < 5) {
       // do something
}
```

"Short Circuited" Operators

If the second part of the expression above were evaluated it would cause errors because of division by zero. The first part of the expression "protects" the second part. If the divisor is equal to zero it is already known that the overall result will be false so the second part of the expression is not even evaluated and the error is avoided.

The conditional AND (&&) operator will only evaluate the second part if the first part is true. The conditional OR (||) will only evaluate the second part if the first part is false.

4.11 Bitwise Operators

- Operate on individual bits of operands
- Bitwise operators are not used very often

	Use	Name	Resulting bit is 1 if...
&	*op1* & *op2*	bitwise AND	both operand bits are 1
|	*op1* | *op2*	bitwise OR	either operand bits are 1
^	*op1* ^ *op2*	exclusive OR	one operand bit is 1, the other is 0
~	*~op1*	complement	operand bit is 0

- 63 & 252 (only last 8 bits shown)

```
        00111111 = 63
&       11111100 = 252
        --------
        00111100 = 60
```

- See notes for complete summary of operation

Bitwise Operators

If used with a floating point type, the bitwise operators will create a compiler error.

Bitwise operations use the same numeric promotion rules as other operations.

The following table demonstrates the results of various bit combinations with each operator:

Operand1	*Operator*	*Operand2*	*Result*
0	**&**	**0**	**0**
0	**&**	**1**	**0**
1	**&**	**0**	**0**
1	**&**	**1**	**1**
0	**|**	**0**	**0**
0	**|**	**1**	**1**
1	**|**	**0**	**1**
1	**|**	**1**	**1**
0	**^**	**0**	**0**
0	**^**	**1**	**1**
1	**^**	**0**	**1**
1	**^**	**1**	**0**

4.12 Shift Operators

- Moves or "shifts" bits of the first operand the distance of the second operand

	Use	Name	"Empty" bits filled with...
<<	*op1* << *op2*	left shift	0
>>	*op1* >> *op2*	right shift	value of highest bit
>>>	*op1* >>> *op2*	unsigned right shift	0

- 236 << 3 (shifts the bits of 236 to the left 3 positions)

```
00000000 00000000 00000000 11101100 = 236

00000000 00000000 00000111 01100000 = 1888
```

Shift Operators

If used with a floating point type, the shift operators will create a compiler error.

The operation 236 << 3 would produce 1888:
00000000 00000000 00000000 11101100 = 236

00000000 00000000 00000111 01100000 = 1888

The operation -236 >> 3 would produce -30:
11111111 11111111 11111111 00010100 = -236

11111111 11111111 11111111 11100010 = -30

The operation -236 >>> 3 would produce 536870882 :
11111111 11111111 11111111 00010100 = -236

00011111 11111111 11111111 11100010 = 536870882

4.13 Overflow And Underflow

- Occurs when the result of a calculation is outside the range which can be represented by a type
- Integer types will be truncated to the number of bits for the type by discarding the high order bits
- Floating point types will store a value of "Infinity" or "-Infinity"

Overflow And Underflow

Floating point calculations that result in "Infinity" values will affect any other calculations which use the resulting value and are usually fairly easy to detect when an error is occurring.

The result of discarding the high order bits for integer types gives a result that is incorrect and closer to zero than the actual result would be but does not otherwise indicate that an error is occurring. Overflows and underflows of integer types are more difficult to detect.

The truncated result of integer overflow or underflow may be positive or negative, depending on the value of the 32nd bit of the result of the calculation. If the 32nd bit is a 1, the result will be negative.

Even though the final result of an expression may be within range for a primitive type, overflow or underflow may still occur depending on the order the expression is evaluated. If a multiplication occurs before a division, for example, the result of the multiplication can cause overflow or underflow before the division occurs. You can control the order of evaluation to avoid such situations as described later in this section.

4.14 Assignment Operators

- Assign new values to variables
- Includes basic assignment op1 = op2
- Includes shorthand operators like op1 += op2 which represents op1 = op1 + op2

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<=
- >>=
- >>>=

- The following pairs of statements are equivalent:

```
count += 5;
count = count + 5;

c *= (a + b) / (c * d);
c = c * ((a + b) / (c * d));
```

Assignment Operators

With the assignment operators, the right hand side is always evaluated first before applying the assignment operator. The right hand side can be any legal expression.

4.15 Ternary Operator

- Most complex of all operators
- Includes a logical (true/false) expression and two other expressions
- Written as:
  - logical_expression ? expr1 : expr2
- If the logical expression evaluates to true, the ternary operator returns the value of the first expression, otherwise it returns the value of the second expression
- The following statement would assign the larger of the two variables myAge and yourAge to the maxAge variable:

```
maxAge = yourAge > myAge ? yourAge : myAge;
```

Ternary Operator

The ternary operator can be used for things like conditional assignment and conditional output.

The following statement would assign the larger of the two variables myAge and yourAge to the maxAge variable:

```
maxAge = yourAge > myAge ? yourAge : myAge;
```

The following statement would output a plural phrase when the value of the variable numBirds is greater than 1:

```
System.out.println("There " + (numBirds > 1 ? "are" : "is") + " " +
          numBirds + " " + (numBirds > 1 ? "geese" : "goose") +
          " in the area.");
```

outputs either:

'There are 4 geese in the area.'

or

'There is 1 goose in the area.'

The ternary operator behaves as an if…else statement but can be more powerful because it can be used as part of a more complex expression.

4.16 Calculation Errors

- Overflow and underflow
- Dividing an integer number by zero produces an exception which halts execution of the program
  - 4 / 0 (halts program)
  - Same behavior for modulo operator (4 % 0)
- Dividing a non-zero floating point number by zero results in Infinity
  - 4.0 / 0.0 = Infinity
- Dividing a floating point zero by zero produces an indeterminate result but does not stop the program
  - 0.0 / 0.0 = NaN ("Not a Number")

Calculation Errors

Integer division by zero is an error which must always be considered when programming. It should be common practice to always "protect" integer division by testing the denominator first to see if it is zero. One way to protect integer division is to use the conditional AND operator to test for zero. The following code:

```
if (total / numPoints > 5)
```

should be rewritten as:

```
if (numPoints != 0 && total / numPoints > 5)
```

If the variable numPoints is zero, the second part of the expression will not be evaluated and will avoid a runtime error.

4.17 Operator Precedence

- Natural precedence
  - Determined by operator precedence groups
    - All operators of higher groups are evaluated before lower groups
  - Affected by operator associativity
- Operators of equal precedence are evaluated left-to-right for left associative and right-to-left for right associative
- Controlling precedence
  - Can be controlled by placing parenthesis around operations to be performed first
  - Required to override natural precedence

Operator Precedence

The following table details the precedence hierarchy of all operators. Operators with higher precedence are listed first.

Group	*Operators*	*Associativity*
postfix	**[], postfix ++, postfix --**	**left**
unary	**unary +, unary -, prefix ++, prefix --, ~, !**	**right**
creation or cast	**(cast), new**	**left**
multiplicative	***, /, %**	**left**
additive	**+, -**	**left**
shift	**<<, >>, >>>**	**left**
relational	**<, <=, >, >=, instanceof**	**left**
equality	**==, !=**	**left**
bitwise AND	**&**	**left**
bitwise exclusive OR	**^**	**left**
bitwise OR	**|**	**left**
conditional AND	**&&**	**left**
conditional OR	**||**	**left**
ternary	**?: (ternary)**	**left**
assignment	**=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, |=, ^=**	**right**

4.18 Precedence Examples

- 5 + 25 / 5 = 10
  - Division performed first
- (5 + 25) / 5 = 6
  - Parenthesis force addition to be performed first
- 10 + 7 / 3 >> 2 = 3
  - Shift is performed last
- c += 3 * 8 % 5
  - 4 is added to the value of c and reassigned to c

Precedence Examples

Although the postfix operators (op++ and op--) are listed at the top of the precedence chart, this only affects the operand which is associated with the operator. For example, the expression:

```
int apples = 4, oranges = 7;
int newTotal = apples+++oranges;
```

is evaluated as:

```
apples++)+oranges
```

instead of:

```
apples+(++oranges)
```

The postfix operator is still evaluated after all other operators in the expression. In the above example, newTotal would be assigned a value of 11 and apples would increment to 5.

Associativity of operators is only considered after all operators of higher precedence have been evaluated. For example:

```
int a = 5, b = -3, c = 12;
a += b *= c / 3;
```

would result in a final value of -7 for a, -12 for b, and 12 for c. The division of c by 3 is carried out first, the '*=' of b second, and '+=' of a last. This is because the associativity of the assignment operators is right-to-left. The following statement is equivalent:

```
int a = 5, b = -3, c = 12;
a += (b *= (c / 3));
```

4.19 Combining Strings

- Two Strings can be combined, or "concatenated" to create a new String
- Performed by using the '+' operator

```
String first = "abc";
String second = "def";
String third = first + second;       // initialized as
"abcdef"
```

- Strings can be concatenated with primitive types which are converted to Strings

```
int one = 1;
String withPrimitives = "This is now " + one + " String"
```

```
// initialized as "This is now 1 String"
```

Combining Strings

Notice in the second example above that the String literals contain spaces. This is to separate the "1" from the other words. Extra spaces often must be added to the beginning or end of String literals to accomplish this spacing. You may also use a String literal which is nothing more than a space character to separate the concatenation of two other values:

```
int numBirds = 5;
String first = "There are";           // notice no surrounding spaces
String second = "geese in the area";  // notice no surrounding spaces
String output = first + " " + numBirds + " " + second;
System.out.println(output);
// Outputs: "There are 5 geese in the area"
```

The number will be converted to a String even if the number comes first in the concatenation.

```
int one = 1;
String message = one + ". Bob Smith";
System.out.println(message);
// Outputs: "1. Bob Smith"
```

Concatenation follows the same rules of operator precedence and associativity. The following code performs the arithmetic addition before the concatenation:

```
int one = 1;
int three = 3;
String precedence = one + three + " results in four";
System.out.println(precedence);
// Outputs: "4 results in four"
```

Concatenation may also be performed with the '+=' operator.

4.20 Coding Tips - Operators

- One space between operators and operands
- If possible, split complex operations into several simpler operations
- Use parenthesis to clarify order of evaluation but only if not excessive

Coding Tips - Operators

Parenthesis may be used to clarify expressions by placing them so they do not affect the order of evaluation of the expression but make the expression easier to understand for others. Any expression which uses more than 3 sets of parenthesis for clarification should be split into several expressions if

possible.

4.21 Control Flow Statements

- Used to alter the flow of a program based on certain conditions
  - Decision statements
    - if
    - if…else
    - switch
  - Iteration statements
    - for
    - while
    - do…while

Control Flow Statements

Most of these control flow statements will be covered later. For now we will cover only the simple decision statements.

4.22 'if' Statement

- Most basic control flow statement
- Executes a statement if a condition is met
- The condition must be a boolean expression
- May execute a single statement or a statement block
- It is possible to reassign variable values within the statement block

```
if (condition)
        statement;

if (condition) {
        statement1;
        statement2;
        statement3;
}

int number = -4;
if (number < 0) {
        System.out.print(
        "negative number");
        number = 0;
```

```
}
```

'if' Statement

It is legal to write an if statement on a single line:

```
if (number % 2 != 0)  System.out.println("odd number.");
```

Although this is legal, it is often best to put the action statement on a separate line from the condition being tested as in the first example on the slide.

The condition of an if statement may be a more complex expression as long as the entire expression evaluates to a single boolean (true/false) value. For example, this statement:

```
if (numOrders != 0 && daysInMonth / numOrders > 4)
         System.out.println("Sales are up from last month.");
```

is legal and desirable to avoid integer division by zero.

4.23 'if…else' Statement

- Extends basic **if** statement
- Executes **else** clause if condition of **if** statement is false
- May be a single statement or a statement block

```
if (condition)
       statement;
else
       statement;

if (condition) {
       statement1;
       statement2;
} else {
       statement1;
       statement2;
}
```

'if…else' Statement

Several if…else statements can be chained together by using another if statement as the body of the else clause. This structure evaluates conditions until one is true, executes the corresponding action statement, and then skips the rest of the if…else statements in the chain. The following statements assign a letter grade based on an integer score:

```
int score = 73;
char grade;
```

```
if (score >= 90) {
          grade = 'A';
} else if (score >= 80) {
          grade = 'B';
} else if (score >= 70) {
          grade = 'C';
} else if (score >= 60) {
          grade = 'D';
} else {
          grade = 'F';
}
```

This code assigns a letter grade of 'C' even though the score would also pass the last condition. This is because the "score >= 60" condition is part of the else clause of the first condition to pass and is never even evaluated.

Care must be taken with this type of expression that logical errors are not introduced. For example, using the following for the beginning of the if…else if statement would assign a letter grade of 'D' even to those who deserved an 'A', 'B' or 'C' because those scores would satisfy the first condition:

```
if (score >= 60) {
          grade = 'D';
```

4.24 Nested Statements

- A second **if** or **if…else** statement may be used within the body of another
- Creates a more complex logical structure
- Use of brackets and indentation even more critical
- Other statements can be nested in a similar fashion

```
if (condition1) {
        if (condition2) {
                statement1;
        } else {
                statement2;
        }
} else {
        if (condition3) {
                statement3;
        }
}
```

Nested Statements

An else clause always belongs to the nearest preceding if that is not in a separate block and not matched with another else clause. Brackets should always be utilized and evaluated to ensure the actual logical structure matches the intent.

Using nested statements is different than chaining statements as discussed on the previous page. With the above example, several conditions will be evaluated before executing any of the statements. To

reach 'statement1', both 'condition1' and 'condition2' must be true. To reach 'statement2', 'condition1' must be true but 'condition2' false. To reach 'statement3', 'condition1' must be false and 'condition3' must be true. If 'condition1' and 'condition3' are both false, no statement is executed.

Rather than creating complex nested statements, the logical operators &, &&, |, and || should be used when possible to combine two or more test conditions into a single boolean expression. This boolean expression may be used as the only condition for a single if or if…else statement. Practice will help you determine when to use nested statements and when to combine conditions using logical operators.

4.25 Coding Tips - if & if-else

- Always use curly brackets to indicate the set of statements that will be executed
  - Even though single statements do not require this it will be easier to add additional statements in the future
  - Code is much easier to read as the brackets clearly delimit the scope of the condition
- Use comments to help document complex combinations of conditions
- Use a unit testing strategy that helps you test every combination of conditions
  - Complex conditional processing is very easy to introduce errors and difficult to identify as the source of bugs
  - Regression testing makes sure these errors are not introduced later
- Consider other ways to do the same thing if conditional statements become too complex

4.26 Summary

- There are lots of different operators available to perform calculations
- Operators can be classified into several related groups
- The type of operand values may be promoted during the course of an operation
- Several errors may occur in calculations which must be accounted for
- Operators have a natural hierarchy which affects the order of evaluation

- Basic decisions about which code path to execute can be made with **if** and **if...else** statements

Chapter 5 - Using Classes and Objects

Objectives

After completing this unit, you will be able to:

- Describe objects, instances, and classes
- Declare variables used to store object references
- Create new objects with initial parameters
- Call methods on object instances

5.1 Objects, Instances, And Classes

- Object - unique from every other object
- Class - common description of a set of objects
- Instance - existing object of a particular class

Objects, Instances, And Classes

Although the words "objects" and "instances" are fairly interchangeable, most of the time objects are described as "instances". This is because in Java there are specific properties associated with an "Object", as we will see later. Using the term "instance" avoids confusion with this definition of an "Object".

5.2 What Are Classes?

- Definition of a kind of object
  - A different class definition for each type of object involved in a solution
- Abstract blueprint for objects
  - Classes do not actually do the work of a program, they merely define the ways in which the actual objects perform work
- Describe fields and methods
  - Contains definitions for common features of all instances

What Are Classes?

Architectural blueprints are a good analogy for class definitions in Java. Blueprints are not the actual building itself but describe everything about a building. A set of blueprints can be used over and over to build several buildings in the same way class definitions may be used to create several instances of the same class.

5.3 Working With Classes And Objects

- Declare the name and type of a variable

```
Video newRelease;
```

  - The class of the object is used as the type of the variable
- Create an object for the variable to refer to

```
newRelease = new Video("Lord of the Rings");
```

  - The 'new' keyword precedes a special type of method called a "constructor". This creates a new instance of the class using the parameters supplied in the constructor

Working With Classes And Objects

Variables which refer to objects are called "reference variables" because they refer to an object of the declared type.

Just as with primitive types, the declaration and initialization of reference variables can be performed on the same line. The two lines from the slide could have been written as one but were separated to demonstrate the two steps involved in creating an instance.

5.4 Working With Classes And Objects

- Use the object's methods to perform work and communicate with other objects

```
newRelease.setPrice(3.99);
System.out.println("The movie title is: " +
       newRelease.getTitle());
```

- To call a method on an object, follow the variable name with a period, the

name of the method, and a comma-separated list of parameters in parenthesis

Working With Classes And Objects

The methods available to any object type depend on the class definition for that type. Later in this unit, we will see various methods of several useful Java classes. In another unit, we will discuss how to define your own classes and methods. If you want to find out the methods available in a standard Java class, you may use the Java API documentation.

5.5 Instantiation

- Creating an object is called “instantiation”
- Classes provide definitions, called constructors, of ways to create instances of a class
  - These constructors are identified using the name of the class
  - Calls to constructors are preceded with the 'new' keyword
  - The parameters passed in when calling a constructor are used to initialize object state
  - Several constructors may be available each with separate lists of parameters

Instantiation

Constructors are called using the format:

```
new className(arg1, arg2, …)
```

The code above creates a new instance of the class and returns the reference to the new instance. This reference must be stored in a reference variable or passed as a parameter to a method. Each of the following examples would make use of the newly created instance:

```
Video rentable = new Video("Fast and Furious");
VideoStore store = new VideoStore();
store.add(new Video("Fast and Furious"));
```

Notice when adding the Video object into the VideoStore it is created "on the fly". The fact that it is being stored in the VideoStore means that it does not need to be referred to locally in the rest of the code. If needed, the instance that was created can be retrieved from the VideoStore.

5.6 Instance Methods

- Execute on a particular instance
- Often change the state of an instance
- Require zero or more parameters
- May return result to location which called the method

```
double total = 0;
total += newRelease.getPrice();
Customer currentCustomer = new Customer("Bob");
currentCustomer.rent(newRelease);
```

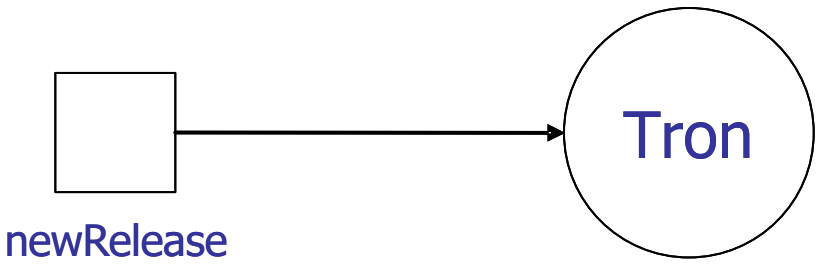
Instance Methods

Many well encapsulated class definitions provide methods for retrieving and modifying the state of an instance. These "get" and "set" methods provide a standard way to work with objects of various types. The convention for these methods is to precede the attribute with 'get' or 'set' as in the example above which retrieves the value of the price of the Video.

5.7 Object References

- Unlike primitive types, variables of an object type do not contain the actual object
  - They refer to the location in memory where the object was created

```
Video newRelease = new Video("Tron");
```



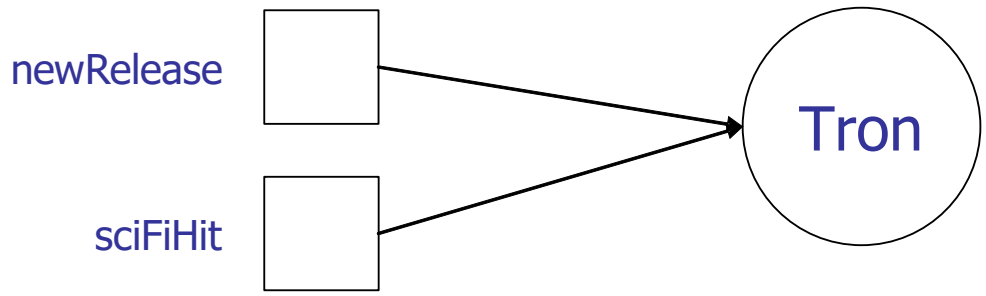
Object References

Java has been said to be a language with "no pointers" and a language of "all pointers". This is because although all object references "point to" the location in memory of an object you may not directly manipulate memory or perform pointer arithmetic as in C or C++. So both of the statements can be considered correct depending on your definition of a "pointer".

5.8 Object References

- More than one variable can refer to the same object
- Using the value of one object reference to initialize another copies only the reference, not the entire object

```
Video newRelease = new Video("Tron");
Video sciFiHit = newRelease;
```



Object References

Object instances are only created when using the 'new' keyword and calling a constructor. The above example initializes two variables to both refer to the same instance. This is because the constructor for the Video class was only called once. Similar code using an int type would result in two variables with two separate values that could be modified without affecting each other.

5.9 Null Values

- If a variable that refers to an object is not initialized, the default value is 'null'
  - This null value means that the variable does not store a reference to an object

- You can also explicitly use the 'null' keyword to set a variable so it does not refer to any object

```
Video familyClassic;  // This stores a 'null'
Video top20Release = null;
```

- The problem with a null value is that if you try to use the variable to call a method on an object while it has a null value that is not allowed
  - How could you call a method to get the title of a Video object when the variable doesn't refer to any object?
  - Trying to call a method on a variable with a null value will cause an error called a 'NullPointerException'

```
Video familyClassic;
familyClassic.getTitle();  // Causes NullPointerException
familyClassic = new Video("Lion King");
familyClassic.getTitle();  // This would be OK
```

5.10 String Operations

- **charAt(int)** – returns character at specified index. First character is at index 0
- **compareTo(String), compareToIgnoreCase(String)** – used to determine the lexicographic order of two Strings
- **equals(String), equalsIgnoreCase(String)** – Determines if the current String and another String contain the same sequence of characters. The first version is case-sensitive the second isn't
- **startsWith(String), endsWith(String)** – determines if a String starts or ends with another String
- **indexOf(…), lastIndexOf(…)** – finds the index of characters or Strings. Various similar methods
- **length()** – returns the length of the String
- **substring(…)** – returns a new String which is a substring. Various similar methods
- **toLowerCase(), toUpperCase()** – returns a new String with all characters in the same case (upper case or lower case)

- **trim()** – returns a new copy of the String with leading and trailing white space omitted

String Operations

Many of these methods return a String object which represents the original String after the operation of the method has been performed on it. Because String objects are immutable, the new String object that is returned must be stored in a variable of type String. The String returned may even be reassigned to the same variable used to refer to the original String object. The following code converts a String to lower case and stores the reference to the new String in the same variable, discarding the old mixed case String:

```
String convertToLowerCase = "Convert Me!";
System.out.println(convertToLowerCase);
          // Outputs: "Convert Me!"
convertToLowerCase = convertToLowerCase.toLowerCase();
System.out.println(convertToLowerCase);
          // Outputs: "convert me!"
```

This is just a sampling of some of the most commonly used methods of the String class. Because the String class is such an important class for the Java language, there are a multitude of methods available. The next lab will show you how to find documentation of all of the methods available for a class.

5.11 "Wrapper" Classes

- Match primitive data types: Boolean, Character, Integer, Double, etc
- Provide an object “wrapped” around a primitive value
- Method types provided:
  - Access the contained value
  - Converting between types
  - Static utility methods to convert between Strings and primitive types
- No 'setValue' type of method
  - Once created, the value of a particular "wrapper" instance is fixed
  - To get another value you have to create another instance

- Common Methods:

```
intValue(), longValue(), …
compareTo(…)
equals(Object)
toString()
```

- Static Utility Methods:

```
parseInt(String)
parseLong(String), …
toString(int)
valueOf(String)
toBinaryString(int)
```

- Constructors:

```
Integer(int i)
Integer(String s)
```

"Wrapper" Classes

All wrapper class types are created by passing in the corresponding primitive type value or a String representation of a value to the constructor.

The wrapper classes combine methods which must be called on specific instances with utility methods defined in the class. The methods in the "Common Methods" section above must be called on a specific instance which contains a value. The "Static Utility Methods" may be called on the class itself.

```
String userInput = getInput();
int i = Integer.parseInt(userInput);
```

Use wrapper objects only if primitive types can not be used. Primitive types will give you the best computational performance and lowest memory usage.

5.12 Autoboxing

- Java 5 introduced a conversion where a primitive value can automatically be converted to a "Wrapper" object and vice versa

- "boxing" is converting a primitive to a wrapper object
- "unboxing" is converting wrapper object to primitive
- "Autoboxing" refers to both and the fact Java does it automatically

- This can include assigning values to primitive or wrapper type variables and use in expressions
- You can't invoke the methods of the wrapper classes on a variable declared as primitive though
- There are "oddities" though when using arrays and '==' tests so it is best to use primitive variables when possible

```
Integer i = 1;     // this is a boxing conversion
int j = i;         // i is unboxed here
i++;               // i is now the same as new Integer(2)
Integer result = i + 2;
int dual = i + result;  // two wrapper objects added
j.intValue();      // This doesn't compile
i.intValue();      // This compiles
```

Autoboxing

One example of an autoboxing "oddity" is the following code:

```
Integer a = 150;
Integer b = 150;
System.out.print(a == b);    // prints false

int x = 150;
int y = 150;
System.out.print(x == y);    // prints true
```

5.13 Summary

- Classes define common state and behavior of a group of objects
- Instances are created from class definitions by use of constructors
- Variables typed with a class are called "reference variables" and refer to an object instance
- Methods are used to access or modify the state of an instance and may require various parameters

- More than one variable may refer to the same instance

Chapter 6 - Writing Classes

Objectives

After completing this unit, you will be able to:

- Provide class definitions to solve unique problems
- Provide definitions for elements of a class including fields and methods
- Describe how and why to "overload" methods
- Describe the difference between local variables and instance variables

6.1 Why Define Your Own Classes?

- Need to use objects tailored to the problem you are trying to solve
- Defining your own classes is a way to provide the abstract definitions of objects that can be instantiated within programs
- List of attributes and functions that will be needed to define the object

Why Define Your Own Classes?

Although the standard Java classes are extensive, they do not define every type of class that is needed to solve a problem. Even if the standard Java classes are enough to hold the data you will require, you must still provide your own instructions and procedures for solving problems.

6.2 Encapsulation

- Controls the ways a class can be accessed or modified
- Hides the implementation of a class from the programs that utilize the class which provides flexibility
- Typically not all class attributes are allowed to be accessed directly
- Rather than providing one monolithic program to handle all possible scenarios, the definitions of different types of objects involved are "encapsulated" and combined to solve a problem

Encapsulation

Encapsulation is a basic concept of Object Oriented programming. With procedural programming, individual components of an entire system rarely make sense when viewed outside of the context of the system. With Object Oriented programming, the idea is to provide smaller component definitions for the various types of objects involved that make sense when viewed separately. When viewed in the context of the system, these definitions take on more meaning as the ways in which these objects are used by other types of objects becomes clear.

6.3 Elements Of A Class

- Fields - the attributes that contain information about the state of an object
- Methods - functions provided to access or modify the state of an object or perform a task
  - Constructors
  - Accessors (get and set)
  - Business methods

Elements Of A Class

Members of a class can be thought of as the individual building blocks of a class. This can be a simplified or very extensive set of definitions depending on the needs of the system.

Constructors will be discussed in a separate section. This section focuses on fields and methods that are not constructors.

6.4 Defining Classes

- Follow the keyword class with the name of the class and a pair of brackets to enclose the rest of the class definition

```
class Video {
       // class definition goes here
}
```

- Comments describing the class may be placed outside of the brackets but all definitions of the elements of a class must be between the brackets

6.5 Coding Tips - Class Definitions

- Provide comments on the general purpose of a class, known bugs, a brief history of the development of a class, and invariant (unchanging) properties of the class
- Indent definitions of elements from the brackets for the containing class
- Add a blank line between each definition of an element of the class
- Provide constructor definitions first, method definitions second, and field definitions at the end of a class definition

Coding Tips - Class Definitions

Comments describing a class should not describe the methods, fields or constructors present in a class. These descriptions should be placed in comments placed immediately before the definition of these elements. In this way the comments will display a similar hierarchy as the class definition itself and allow others to easily access the documentation of the class at the appropriate level of detail.

6.6 Fields

- Variables used to store data within the class
- These may be variables of primitive or reference types
- A field can refer to another object and can even refer to another instance of the same class

Fields

All the data associated with a particular instance of a class must be stored in fields. The various fields of an instance define the "state" of the object which may be different from the state of other instances of the same class. In a well encapsulated class, the values of these fields are not directly accessible outside of the class but must be accessed using various methods. The mechanism used to protect fields will be discussed later.

6.7 Defining Fields

- Declared like other variables by declaring the type and name of the field
- May be initialized when declared or deferred for constructors to initialize

```
class Video {
       String title;
       String category = "New Release";
}
```

- Also called instance variables because they hold unique values for each instance of a class

Defining Fields

The fields of a class are declared within the brackets for the class definition but not within any methods.

In the example from the slide, one field is initialized when declared while the other is not. This is because it makes sense to have a default value for the category, in this case "New Release", while there is no default value for the title of a Video. The title of a Video must be initialized by a constructor. A constructor may also change the value for the category or may leave the default value.

6.8 Coding Tips - Fields

- Same naming conventions as used with other variables
- Field comments should describe the field's purpose, give an example, and point out if the field is accessed concurrently or visible to outside classes
- Methods provided to access and modify fields should be named with the prefixes 'get' and 'set' before the name of the field
  - These methods should always be used to access fields, even from within the class itself
  - If a field is a boolean type, the method to access the value should be 'is…'

Coding Tips - Fields

Simply because fields are primitive types with descriptive names is not a reason to omit comments describing a field. Comments should be written so that someone unfamiliar with the program or even the problem domain can understand the purpose of the code the comment describes.

6.9 Methods

- A reusable block of code that performs a specific task
- May take various types as parameters, or input, to the method
- Returns either no value or a single value
- May return a primitive or reference type
- Defined within a class definition but separately from other elements of the class

Methods

The number and types of methods you define in your own classes will depend greatly on how the class is intended to be used. Besides the methods that are used specifically in your program, thought should be given to methods that may become useful in the future.

The combination of the method's name and the parameters for the method form the method's "signature". The return type of a method is not part of the signature.

6.10 Defining Methods

- Method definitions must contain the type of value returned from the method, a method name and a list of parameters in parenthesis (empty if no parameters)
- If no value is returned from a method, the keyword void must be used in place of the return type
- The **return** keyword precedes the value returned by the method, which is usually the value of a variable

```
class MyClass {
	returnType methodName(arg1, arg2, …) {
		// method body
	}
}
class Customer {
	Address createAddress(String street, int zip, ...) {
		// method body
	}
```

```
        void updateAddress(Address newAddress) {
                // method body
        }
}
```

Defining Methods

Once a return statement is reached, the method halts execution and returns the value indicated. This return value can be used in the code which called the method. If there is no return statement, because the return type is labeled as void, the methods executes until reaching the last statement in the method definition and then returns to where the method was called from.

When declaring parameters, a type and name must be declared just as with other variables. Initial values are not declared because these are supplied by the code calling the method. Parameters become variables available within the body of the method.

Some sample methods for the Video class:

```
class Video {
          String title;
          String getTitle() {
                    return title;
          }
          void setTitle(String newTitle) {
                    title = newTitle;
          }
}
```

Notice the getTitle() method has a return statement while the setTitle(String) method has a void return type.

6.11 Passing Parameters

- When parameters are passed into methods, they are passed "by value"
- This means that a copy of the value is created when passed into the method
- For primitive values, a method can't change the original value from the location where the method was called
- For objects, because the original variable only refers to the actual object, copying that reference still allows the method to access the original object

Passing Parameters

Because passing in a reference to an object may allow a method to alter that object, any such behavior should be well documented in comments relating to the method. This behavior must be documented so current and future developers are aware of these behaviors.

The only way to change the value of a primitive type variable by calling a method is to use the value returned by the method to re-initialize the variable. The following code uses the function from the Math class which returns the square root of the value passed as a parameter:

```
double number = 16.0;
number = Math.sqrt(number);       // value stored is now 4.0
```

When calling the method, the original value of 16.0 is passed in as a parameter. After the method executes, the variable contains a value of 4.0.

6.12 Overloading Methods

- Providing multiple methods with the same name but different lists of parameters creating different “method signatures”
- Used to provide multiple functions which produce similar effects but use different input
- Allows the creation of logical "groups" of methods which are similar but still defined separately

```
void print(Account account) {...}
void print(Account account, String message) {...}
```

- Method declarations are not allowed to only differ by the return type of the method
  - The method name or list and order of parameters must always be different

Overloading Methods

Method overloading is extremely useful because it would be very tedious and unintuitive if every method defined had to have a separate name. Every method must have a different signature from every other method but this can be achieved by providing different lists of parameters.

The specific order of the parameter types is also taken into consideration when determining a method's signature.

When calling a method, the types and order of parameters determines which of the overloaded methods is executed.

6.13 Coding Tips - Methods

- Parameters follow the same naming conventions as other variables
- Method comments describe what a method does and why, a comment for each parameter, the method's return value, any exceptions thrown, the visibility of the method, pre- and post-conditions which apply, and in what ways the method changes the instance
- Parameter comments describe the use of the parameter and any pre-conditions which must be applied to the parameter value
- When overloading methods, avoid methods which differ only in the order of parameter types. Although legal, this is unneeded confusion

Coding Tips - Methods

Pre- and post-conditions are statements that are true before or after a method executes. Often they deal with acceptable values for parameters and the state of the object after a method is called. These conditions also describe various assumptions made regarding a method.

6.14 Local Variables vs. Instance Variables

- Instance variables are available to all methods in a class
  - Although available, instance variables should only be accessed or modified by calling the methods to 'get' and 'set' these values
- Local variables are declared within a method and are only accessible within that method
- Parameters from the method declaration become local variables within the method

Local Variables vs. Instance Variables

Local variables are often declared within a method to hold temporary results of calculations. This is preferred to reusing some variable declared as a parameter as these values may need to be used more than once in various parts of a method. A local variable is often declared at the beginning of a method to hold the value returned by the method.

6.15 Example - Defining a Class

- Below is a simple definition of an "Account" class

```
public class Account {
	String firstName;
	String lastName;
	double balance;
	public void print() {}
}
```

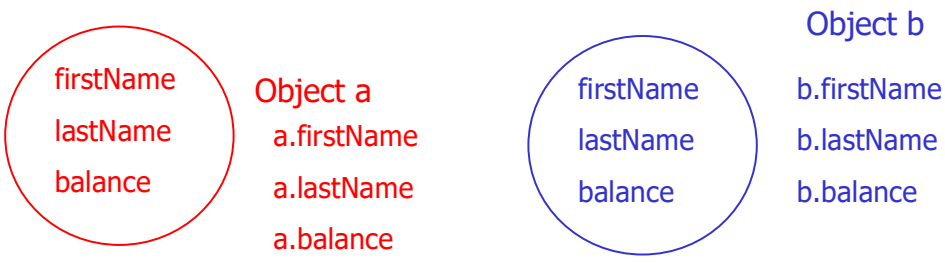
Example - Defining a Class

Above is a user defined data type named Account.

The fields and methods that compose a class are often called its members.

6.16 Example - Fields

```
public class Driver {
	public static void main (String args[]) {
		Account a = new Account();
		Account b = new Account();
	}
}
```



Example - Fields

Instance Member – data associated with a particular object. Each instance of a class has its own copy of the instance member defined in the class.

When object a is created, a copy of the instance members firstName, lastName and balance are created with a default value of null, null and zero, respectively.

When another object b is created, a copy of the instance member firstName, lastName and balance is also created.

The firstName in object a is independent from the firstName is object b.

6.17 Example - Fields

```
public class Driver {
	public static void main (String args []){
		Account a = new Account();		(1)
		a.firstName = "Bill";			(2)
		System.out.println(a.firstName);	(3)

		Account b = new Account();		(4)
		b.firstName = "Jane";			(5)
		System.out.println(b.firstName);	(6)
	}
}
```

Example - Fields

In statement number:

(1) An instance of Account is created. The keyword **new** is used to create an instance of the class. In this step, the member variable is initialized with its default value and the constructor is called.

(2) The name "**Bill**" is assigned to the member **firstName** of object **a**.

(3) Prints out the value of **a.firstName**. (which is "Bill")

(4) Another instance of Account is created.

(5) The name "**Jane**" is assigned to the member **firstName** of object **b**.

(6) Prints out the value of **b.firstName**. (which is "Jane")

6.18 Example - Defining a Method

- The following code defines the 'print' method

```
public class Account {
```

```
        /* fields omitted from sample */

        void print() {
                System.out.print(firstName + " ");
                System.out.print(lastName + " ");
                System.out.println("$ " + balance);
                return;       // optional
        }
}
```

Example - Defining a Method

In the class Account, a method **void print()** is added. Methods are declared in the class, but outside any main method that may be present (note: the main() is a method itself).

The method header consists of the following parts:

(1) Return Type - this is the type of value the method returns. In this given example there is no return type, so void is used.

(2) Method Name - the name of this method which is print.

(3) Parameter - a type and name should be given to the parameter. In this given example, there is no parameter so an empty parenthesis is used.

Return Value - if a method has to return a value, the actual value (corresponding to the return type) must be passed back using the return keyword. If there is no return value, it is optional.

The method body is the set of statements between the curly braces {}.

6.19 Example - Calling a Method

- The following code creates an Account and calls the 'print' method

```
public class Driver {
        public static void main (String args[]){
                Account a = new Account();
                a.firstName = "Bill";

                a.print();
        }
}
```

- The value of 'firstName' and 'lastName' printed by the 'print()' method is

determined by what object the method is called on

Example - Calling a Method

The main method is the place where the application performs its tasks.

In the code, an instance of an Account is created.

To call an instance method, the syntax is:

```
object.method();
a.print();
```

6.20 Summary

- Custom classes are defined to solve specific problems and model unique object types
- Fields, methods, and constructors are all elements of a class
- Fields contain data related to a class
- Constructors provide ways to create instances of a class
- Methods provide ways to interact with objects
- Multiple methods with the same name but different parameters are said to be "overloaded"

Chapter 7 - Controlling Code Access and Code Organization

Objectives

After completing this unit, you will be able to:

- Understand how access modifiers enable encapsulation
- Organize code into packages
- Import existing code from user-defined or Java standard packages

7.1 Controlling Access

- Providing direct access to data is not a good idea

```
myAccount.balance -= amountWithdrawn;
```

- In the above example the Account object has no way to control what happens if the account would be overdrawn
- Access modifiers can be used to control access to elements of a class
- **public** – Can be accessed from anywhere, including within other classes
- **private** – Can only be accessed from within the same class
- These keywords can be added to the beginning of a field, constructor, or method definition

Controlling Access

Public classes must be declared in a source file of the same name. A public BankAccount class must be declared in a 'BankAccount.java' file.

Although any access modifier may be used with any definition mentioned above, some combinations are more common than others.

Classes are most often declared as public. This allows them to be used from within any other class.

Constructors are most often declared as public. This allows instances of the class to be created from within any other class.

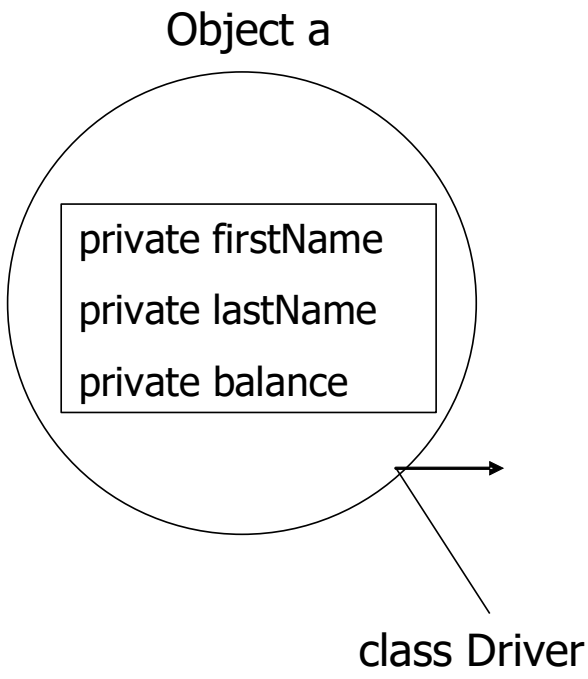
Fields are most often declared as private. This prevents direct access from other classes and forces other classes to use the methods defined within the class to access or modify the value of a field.

Methods are most often declared public but certain methods may be declared private. Any methods

that are defined simply to simplify other methods by removing repeatable blocks of code may be declared private if it does not make sense to call such a method from another class.

Other access modifiers are available and will be discussed later.

7.2 Data Hiding



- Use the keyword private in the declaration of the instance variables
- The variable is only accessible within the class
- It forces code in other classes to use methods to modify the member variables

Data Hiding

```
public class Account {
          private String firstName;
          private String lastName;
          private String balance;
          public void print() {
                    System.out.print(this.firstName + " ");
                    System.out.print(this.lastName + " ");
                    System.out.println("$ " + this.balance);
          }
}

public class Driver {
          public static void main (String args []){
                    Account a = new Account();
                    a.firstName = "Bill";      // error!
          }
```

```
}
```

7.3 Encapsulation

- Hides the details of a type's implementation
- Other classes interact with the type through the exposed behavior (method), not directly with the internal implementation
- This makes the class more maintainable

Encapsulation

In the class Account, add getter/setter of the instance variables to modify and access the private variables of the class.

Getters – you want to read the content of the variable (accessor).

Setters – you want to modify the content of the variable (mutator).

```
public class Account {
          private String firstName;
          public String getFirstName() {
                    return firstName; }
          public void setFirstName(String fname) {
                    firstName = fname;           }
}
public class Driver {
          public static void main ( String args [] ) {
                    Account a = new Account();
                    a.setFirstName("Bill");
                    System.out.println(a.getFirstName());
          }
}
```

7.4 JavaBeans

- JavaBeans is a common term used to refer to well-encapsulated classes
- JavaBeans typically provide
  - get/set methods for accessing data
  - Methods where the name of the method intuitively defines the function
- JavaBeans are often used to transfer data between application components

- It is much easier to transfer one object with all the related data than each piece of data separately

JavaBeans

The Account class that has been used in many examples is an example of a JavaBean.

Imagine how difficult it would be to write a program if rather than the Account class definition we had to maintain separate references to each element of data individually. The Account class provides an "encapsulation" or packaging of this data.

Technically the JavaBeans specification is more formal than defined above. It was originally defined to enable visual programming tools to be able to interact with Java classes within a visual design environment. The term has been used much more loosely since the concept of encapsulation applies to much more than just visual programming tools.

7.5 Packages

- Are a collection of functionally related classes
- Provide access protection and namespace management
- Provides the ability to organize large numbers of classes
- Are a part of the Java language

7.6 Naming Packages

- Common to start with reversed Internet domain name
- Append internal department or organization if applicable
- Append application or layer name
- Finally add descriptive name of package
- Parts of a package name are separated by periods

```
com.webage.payroll.employee
```

Naming Packages

Although the first two sections of the above package name could be left off and a descriptive package of 'payroll.employee' would still be left, this is not recommended. If this code was integrated in the

future with code from another organization, which may have their own 'payroll.employee' package, all of the code would need to be repackaged. Since this is a difficult process prone to errors, any steps that can be taken to avoid this are beneficial.

Unlike class names, package names are almost always lowercase. Package names are also generally one word long. Because package names are meant to be general descriptions of the contents, specific names with more than one word typically limit the generality of the package name and might need to be changed in the future.

7.7 Declaring Packages In Classes

- Use **package** statement to declare package membership
- Must be first statement in source code
  - May be preceded by comments

```
/* This comment is legal
before package statement */

package com.webage.payroll.employee;

public class Manager {
…
```

7.8 Problems Solved With Packages

- Common names used by different people
  - The common class name may be kept but a package used to differentiate the classes
- The large number of classes required to solve complex problems become unmanageable
  - Using packages allows for the creation of organizational units
- Public and private scope does not allow for proper limiting of scope
  - Packages allow for a scope which is in-between public and private

7.9 Package Access

- Packages add to the list of access modifiers and change the behavior of others
  - **public** - Accessible to any class from any package
  - **private** – Only accessible within the class
  - **protected** – Accessible from other classes within the same package
  - No access attribute - Accessible only from other classes within the same package

Package Access

When an access modifier is not present this is often called "default" access.

The public and private access modifiers have been mentioned previously. This slide represents all four access modifiers that may be used with classes, methods, instance variables or class variables. According to this slide the "protected" and "default" levels of access appear to be no different. The addition of inheritance in another unit will differentiate these two levels of access in relation to subclasses.

As mentioned previous in the course, a public class must be saved using the same filename as the class name. Therefore, each source code file can have at most one public class.

7.10 Example - Access Modifiers

```
public class Account {
	private String firstName;
	protected String lastName;
}

public class Driver {
	public static void main (String args[]) {
		Account a = new Account("Bill", "Joy");
		System.out.println(a.firstName);// error!
		System.out.println(a.lastName);// OK
	}
}
```

Example - Access Modifiers

The variable **firstName** is **private**. It is not visible outside the class.

The variable **lastName** is **protected**. It is accessible in the class Driver, provided that they are in the same package.

7.11 Import Statement

- By default, the fully qualified class name, including package, must be used to refer to the class
  - This can be cumbersome and prone to mistakes
- Addition of an **import** statement allows the use of simple class names without a package qualifier
- Must be placed after any package statement and before other statements
- You can import a single class or an entire package

```
import com.webage.payroll.employee.Manager;
// import single class
import com.webage.payroll.system.*;
// imports all classes
```

Import Statement

Adding an import statement does not add extra code to a class. It simply allows the compiler to understand simple class names and find the definition of the imported class.

Importing an entire package does not import any sub-packages, only all classes located in the imported package. To import sub-packages, you must add an import statement specifically for the sub-package.

```
import package.*;
import package.subpackage.*;
```

7.12 Using Classes From Packages

- When referring to classes, there are two options:
  - Use fully qualified name
  - Use "simple" class name without package identifier and use import

statements and package contents to resolve reference

```
package com.webage.payroll.system;
import com.webage.payroll.employee.Manager;
public class Timecard {
   private Manager supervisor;     // simple name OK
   private com.webage.payroll.employee.Employee owner;
   //fully qualified name
...
}
```

Using Classes From Packages

If the name of an imported class is the same as the name of a class in the package, the fully qualified name must always be used. If such a conflict exists and a simple name is used a compilation error will occur.

7.13 Coding Tips - Import Statements

- Typically organized and grouped by packages
- Importing entire package typically avoided
  - This could lead to naming conflicts in the future if classes are added to the imported package

```
import com.webage.payroll.employee.Employee;
import com.webage.payroll.employee.Manager;

import com.webage.payroll.system.Timecard;
import com.webage.payroll.system.WorkDay;
```

Coding Tips - Import Statements

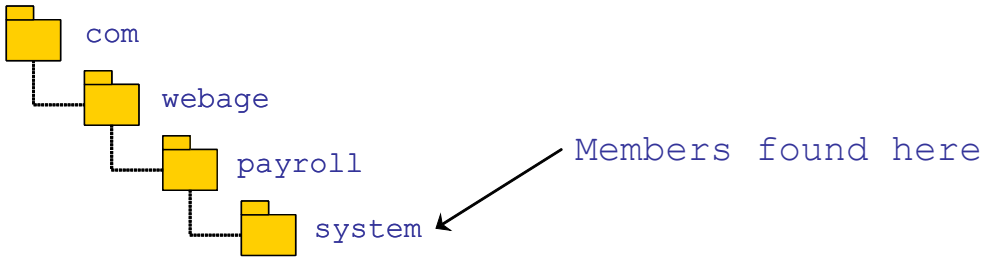
The statement above that importing entire packages is "typically" avoided is somewhat ambiguous because this practice is one that does not clearly have more benefits using one standard as opposed to the other. Using explicitly imported classes will avoid naming conflicts but will lead to more work when maintaining the list of imports if a class needs to be imported in the future. By importing entire packages, this maintenance work may be reduced but lead to name conflicts in the future. Importing entire packages may also lead to some import statements which must go against the standard to import specific classes to avoid such name conflicts.

This topic is one of the most hotly debated coding standard topics within organizations. It is ultimately up to the organization to establish their own standard based on evaluation of the benefits and drawbacks of each practice outlined above.

Many Java development tools will help you maintain the list of import statements, even adding import statements for you to qualify simple class names used in code. Although this may be the case, it is still a good practice to formulate a standard for the use of import statements.

7.14 Correlation To File Structure

- Package organization within code must match file organization within file structure
- Package members must be placed in deepest directory
  - Members of the com.webage.payroll.system package would be found in the com/webage/payroll/system directory



Correlation To File Structure

The requirements for package directories is similar to that for filenames. A public Employee class that is part of the com.webage.payroll.employee package must be defined in an Employee.java source file located in the com/webage/payroll/employee directory. If this is not the case, compilation errors will occur.

It is up to you to place files in the correct directories. The Java tools will not check files when you save them, only when they are compiled.

The compiled class files may be placed in the same directory as the source code or a different directory. If placed in a different directory, the file structure must also match the package structure.

7.15 Class Path

- Used to indicate where to search for Java types
- Points to top level folder of package hierarchy
- Uses platform dependent path separator character if indicating multiple paths
- Set using '–classpath' command line option or CLASSPATH environment variable

```
C:> javac –classpath "C:\Java Code" Manager.java

C:> set CLASSPATH=.;"C:\Java Code"
C:> javac Manager.java
```

Class Path

In the above example, the directory is enclosed with quotes because it contains a space. Directories which do not contain spaces do not need quotes.

Multiple entries on the path are separated by the platform-specific path separator. This is a semicolon on Windows and a colon on Solaris and Linux.

By default, the class path is the current directory. If the –classpath option or the CLASSPATH environment variable is set, this default is overridden. If you still want the current directory to be part of the class path, you can include a period as one of the entries in the list of paths using either method. This is demonstrated above with the command which sets the CLASSPATH environment variable.

When possible, the command line option is the preferred way to set the class path. This is because you can modify the class path for every execution of the compiler or interpreter. The command line option overrides the environment variable if it is set.

The class path only has to point to user-defined classes. The standard Java classes will automatically be found if imported into other classes.

7.16 Java Core Packages

- java.lang - fundamental classes of Java language
  - Available without imports
- java.util - provides utility and collections classes
- java.io, java.nio - provides for system input and output
- java.security, javax.security - built-in security mechanisms for Java
- java.awt, javax.swing - classes for user interfaces and painting graphics and images
- java.text - provides for handling text, dates, and numbers
- java.net - provides classes for network communication
- javax.print - provides for the Java Print Service

Java Core Packages

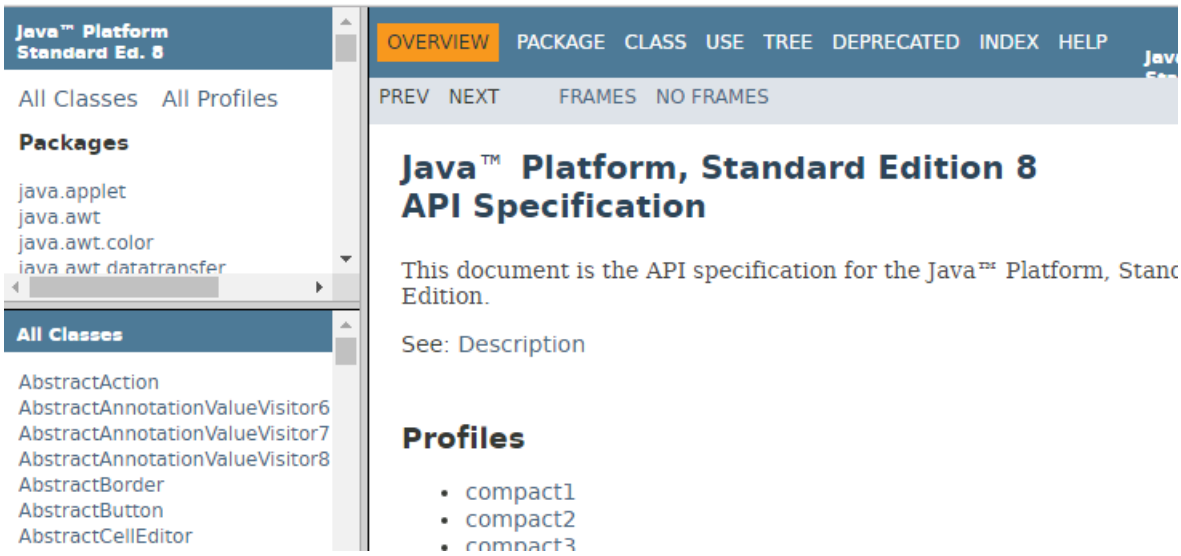
Any of the standard Java packages or classes are imported in exactly the same way as user-defined classes.

These are just a few of the hundreds of packages available in the Java language.

Many of these packages also have several subpackages but the highest level package is the best general division of the contained classes.

7.17 Java API Documentation

- Detailed documentation of all Java classes
- Invaluable tool when programming
- Available online at:
  - Java SE 7 – http://docs.oracle.com/javase/7/docs/api/
  - Java SE 8 - http://docs.oracle.com/javase/8/docs/api/
- You can also open from within Eclipse by selecting a class from the *Hierarchy* view or in the source editor and selecting **Navigate → Open Attached Javadoc**



Java API Documentation

The Java API documentation is a set of web pages that detail the various packages and classes of the Java language. It is not installed as part of the software development kit but can be downloaded and installed separately. The API documentation is also available online at:

http://docs.oracle.com/javase/8/docs/api/ (browsing online)

http://www.oracle.com/technetwork/java/javase/downloads/index.html (downloading separately)

Pick the version of the documentation that matches the version of Java you are using. The documentation above shows Java SE 6.

7.18 Summary

- Using access modifiers is critical to writing well-encapsulated code
- Encapsulation helps increase the modular nature of Java code
- Packages can be used to help organize code
- Use the Java API documentation to find out what is already included in Java

Chapter 8 - Constructors and Class Members

Objectives

After completing this unit, you will be able to:

- Define Constructors for a class
- Call constructors with initial parameters
- Define Class members
- Compare Class members with Instance members

8.1 Constructors

- Are special methods which have the same name as the class
- They have no return type:
  - This is because the implicit return type of the constructor is the class type itself
- This method gets called when an instance of the class is created
  - Invoked using the new keyword
- Can accept parameters to initialize instance variables
- Constructors are generally public methods

Constructors

The **new** keyword is a call to the constructor of the type being referenced:

```
Account a = new Account();
```

A protected constructor would prevent an object of the class being created except from code that is either in the same package or a subclass.

A private constructor would only allow the object to be created from within the code of the same class. Although this sounds like it would prevent any objects being created an object could be constructed in a 'static' method within the class. 'Static' methods are described later in this chapter.

8.2 Default Constructor

- In the absence of other constructors, the compiler will supply the "default"

constructor

- If you start defining your own constructors, you will lose the default constructor supplied by the compiler

- A constructor with no arguments and has an empty body
- Enables you to create an object calling **new className()**

8.3 Multiple Constructors

- Multiple constructors can be defined with different lists of parameters and used to create objects in different ways
  - The name of the constructor will always match the class name
- If no constructor is defined, a default constructor is supplied by the compiler
  - This constructor has no parameters
  - Fields are initialized with the default value for the type unless an initial value is supplied with the field definition
  - If **any** constructor is defined, the compiler will not supply the default constructor

Multiple Constructors

Depending on the type of object you are defining, the number and type of constructors will vary. For some objects, having a no-argument constructor would not make sense if there is some minimum amount of information that must be supplied to create an object. There may be default values for some or all of the fields of a class so multiple constructors, even a constructor with no arguments, may be appropriate.

8.4 Defining Constructors

- The name of a constructor must match the class name exactly
- Unlike a method, no return type is specified
- Parameters are declared the same as other methods

```
class Video {
```

```
        public Video (String newTitle) {
                title = newTitle;
        }
        public Video (String newTitle, String newCategory) {
                title = newTitle;
                category = newCategory;
        }
        // Fields and get/set methods omitted
}
```

Defining Constructors

Although no return type is given in a constructor definition, you can think of the constructor returning a new instance of the class initialized with the values of the parameters.

8.5 Example - Calling Constructors

- Using the Video class described previously

```
public class Driver {
  public static void main(String[] args) {
    Video aVideo = new Video("Tron", "Sci-fi");
    // The 'aVideo' variable now refers to a Video object

    VideoStore store = new VideoStore();
    store.add(aVideo);
    // The VideoStore now contains a reference to
    //the video  also
  }
}
```

Example - Calling Constructors

The 'VideoStore' class is a simple class to track videos that are available to rent or rented. The VideoStore class would need a "default" constructor since we are creating a new VideoStore in the 'main' method.

8.6 "Good" Constructors

- There is not really any rule that describes if a constructor is "good" or not
- The constructors that are needed will depend on in what ways objects of that class could be constructed
- You can ask a few questions to help determine what constructors a class might have
  - Are there any properties that an object of the class must always have set? These will be properties that would be present in the parameters of every defined constructor
    - Example: An Account can't be created without an ID and customer name
  - Are there properties that might optionally be supplied but if not supplied could be set to some default value? These properties may be present in the parameters of some constructors but not in others where the default value would be set
    - Example: An Account could be created by passing in an initial balance but if not supplied the balance is set to 0

"Good" Constructors

In the example above the account ID might not always be a parameter in the constructor. Perhaps this is a property that is automatically generated internally when the Account object is created to guarantee uniqueness. Parameters that will be present in every constructor are properties that must always be set and will always be supplied by the code calling the constructor to create the object.

8.7 'this' Keyword

- The keyword 'this' refers to the current object
- There are two main reasons you may use the keyword this
  - The name of a parameter in a constructor is the same as a field that is to be set
    - Without 'this' you would be modifying the value of the parameter instead of the field

```
public Account(String firstName) {
        this.firstName = firstName;
}
```

- Calling one constructor from another

```
public Account(String firstName, String lastName) {
        this(firstName);
        ...
}
```

8.8 Using 'this' to Call a Constructor

- The keyword 'this' is used to call a constructor within the class.
- Creates good reuse of existing constructors

```
public class Account {
  public Account(String fName, String lName, double bal) {
       firstName = fName;
       lastName = lName;
       balance = bal;
  }
  public Account(String fName, String lName) {
       this(fName, lName, 0);
  }
  public Account (String fName) {
       this(fName, "", 0);
  }
  public Account() {
       this("", "", 0);
  }
  // Fields and get/set methods omitted
}
```

Using 'this' to Call a Constructor

In the class Account, the constructors are overloaded with the following signatures:

```
public Account (String fName, String lName, double bal)
public Account (String fName, String lName)
public Account (String fName)
public Account ()
```

Therefore, there are lots of ways to create an Account object.

If we wanted to require that objects always contain certain data we could simply not declare constructors that allow objects to be created without that data.

In the statement:

```
Account b = new Account (“Bill”);
```

The constructor whose signature is Account (String fName, String lName, double bal) is invoked. The value of "Bill" is passed in for the first name, the empty String "" is passed in for the last name, and the Account is created with a zero balance.

8.9 Using 'this' to Set a Field

- Required if the name of the parameter for the constructor is the same as the field

```
public class Video {
	public Video (String title) {
		this.title = title;
	}

	private String title;
}
```

- Generally it is best to avoid doing this if possible
  - Give the constructor parameter a different name to indicate the purpose

```
public Video(String initTitle) {
	title = initTitle;
}
```

Using 'this' to Set a Field

Even though the field above is private, it can be accessed directly since the constructor is part of the class.

Using a unique name for the constructor parameter will help avoid a situation where the 'this' keyword is mistakenly left out and the constructor does not actually set the value of the field. This is a difficult error to debug.

8.10 Class Members

- Class members belong to the class itself
- Class members are shared by every instance of a class
- Class members are available even if no instances of the class exist
- Class members are declared using the **static** keyword

Class Members

You may run across the term "members of a class". This term is sometimes used to refer to the individual definitions contained within a class. Because this term can refer to instance members or class members, it is intentionally avoided to prevent confusion. The term "elements of a class" is used to refer to these definitions.

8.11 Examples Of Class Members

```
public class Video {
	private static int numVideos = 0;

	public static int getNumVideos() {
		return numVideos;
	}

	public Video(...) {
		numVideos++;
		...
	}
}

	// Other code
	Video.getNumVideos();
```

Examples Of Class Members

Since class methods are declared by using the keyword static, they are often referred to as "static methods". Although class variables are sometimes referred to as "static variables" or "static fields", these terms are not used as often as they would imply that the value of these variables is fixed. This is not the case with class variables and we will see later how to declare variables with a fixed, or constant,

value.

8.12 Comparison With Instance Members

- Instance variables are created when an instance of the class is created
- Instance methods are called on a particular instance of the class
  - Only instance methods may refer to instance variables
  - Instance methods may refer to class variables or class methods
- Class variables are created when the class is loaded. This happens when:
  - An instance of the class is created
  - The class is referred to in a program
- Class methods are called on the class itself
  - Can be called even if no instance of the class exists

Comparison With Instance Members

The reason only instance methods may access instance variables is because of the manner in which class methods may be called. Since class methods may be called even before instances of the class exist, they may not refer to instance variables which would not exist. If instances of the class did exist, there would be no way to determine which instance variables to refer to if the method was called on the class itself.

This restriction also includes instance methods. Instance methods may not be called from within class methods unless an instance of the class is created and the instance method is called on that specific instance.

8.13 Use Of Class Variables

- Used for values that are part of the entire set of instances rather than one particular instance
- Something that describes the entire group
- Number of instances which exist is one example
- In reality static class variables are not as common because data is usually

associated with an instance of a class

- ◇ You could use static class variables as "global" variables but this is discouraged in Java as it is not object oriented

8.14 Static Class Methods

- A static method can be invoked without any instance of the class to which it belongs
- There is no "this"
- Inside this method, only static (and local) variables are accessible. Attempts to access non-static variables (instance variables) will cause a compiler error

Static Class Methods

The best example of the static method is the main() method:

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

The main() method is static because it has to be accessible in order for an application to be run, before any instantiation can take place.

In the package java.lang, open the class Math. You will see a list of methods that are static. To use these static methods, there is no need for you to create an instance of class Math. You can invoke the method using the class name as shown below:

```
public class UseMath {
          public static void main (String args[]) {
                    int number = -98;
                    System.out.println(Math.abs(number));
          }
}
```

8.15 Use Of Class Methods

- Main methods to execute programs
  - ◇ This special method describes the steps that occur within the program
- Utility functions
  - ◇ Methods that perform a specific task or calculation where the result is returned from the method and does not need to be remembered as

"state". Accepts all needed information as parameters

Use Of Class Methods

Although we have discussed the special main(String[]) method before, it will be discussed further in the next slide.

Because a large number of utility classes and methods are provided by the standard Java classes, the need for class members is generally not as prevalent as the need for instance members within user-defined classes. However, there may be specific business processes or functions which would benefit from the definition of your own utility classes and/or methods.

8.16 The Math Class

- Special class for performing mathematical functions
- Special "static" methods can be called directly on the class itself instead of an instance

```
double result =
Math.sqrt(4.0);
```

- Common Methods:

sin(double), asin(double)

cos(doulbe), acos(double)

tan(double), atan(double)

toDegrees(double)

toRadians(double)

abs(…), pow(double, double)

log(double), exp(double)

ceil(double), floor(double)

max(…), min(…)

random(), sqrt(double)

The Math Class

In general, static methods are utilities like those available in the Math class who take all information as parameters and return the result of some operation. Static methods will be discussed in more detail later.

The max(…) and min(…) methods above come in many different forms. All forms take two parameters but take different primitive types. The two parameters always match in type and the result returned is the same type as the parameters.

The random() method above returns a pseudorandom double value greater than or equal to 0.0 and less

than 1.0. This number is referred to as pseudorandom because it is not possible to model a true random process, say throwing a die, with a mathematical algorithm. This method returns values in the range with an approximately uniform distribution.

To obtain a positive random number in some range, the result of calling the random() method may be multiplied by the maximum number in the range.

8.17 Main Method And Command Line Arguments

- The **main(String[])** method is a special static method used to execute a program
- Always declared with the same signature

```
public class RentalSystem {
	public static void main(String[] args) {
		// statements to execute program
	}
}
```

- Any additional command line arguments are initialized as values of the String array

Main Method And Command Line Arguments

Using command line arguments is a good way to provide information to a program at runtime. The following code simply prints out the command line arguments but demonstrates the process:

```
public class MainDemo {
	public static void main(String[] args) {
		for (int i = 0; i < args.length; i++) {
			System.out.println(args[i]);
		}
	}
}
```

The following shows the command given and the output produced:

```
>java MainDemo print these out
print
these
out
```

8.18 Declaring Constants

- The keyword **final** indicates that the value of a variable may not change
- Often used for a class variable to declare a constant for the class
  - Uses the static and final keywords together to declare the constant

```
public class CreditCard {
       public static final double APR = 0.26;
```

  - Many standard Java classes have useful constants
  - These constants are used by following the name of the class with a period and the name of the constant

```
double circumference = Math.PI * 2 * radius;
```

Declaring Constants

Although the use of constants is more frequent with class variables, it is also possible with instance variables. Since most instance variables represent changeable state of an instance, this use is not as common. Some instance variables may make more sense as constant values, however. The title of a Video might be one example of a variable that would not change once initialized.

8.19 Coding Tips - Class Members

- Usually grouped together and separated from instance members by white space
- Names of constant variables in all capitals with underscores separating words

```
public static final int MAX_VIDEOS = 1000;
```

Coding Tips - Class Members

To provide maximum flexibility in coding, it may be best to retrieve values of constants using methods rather than direct field names. This would allow for future change in names of constants and provide for a way to calculate values rather than direct initialization of constants.

8.20 Useful Standard Class Members

- Math class
  - Various mathematical functions available as static methods
  - Constants for Pi and e (natural logarithm base)
- "Wrapper" classes
  - Various methods for comparing and parsing primitive types and Strings
  - Constants for maximum and minimum values

8.21 Summary

- Constructors describe the ways to create objects of a class
- Classes contain a "default" constructor unless you define unique constructors
- Class members, variables and methods defined with the 'static' keyword are shared by all class instances
- The 'final' keyword indicates something may not change once it is set

Chapter 9 - Advanced Control Structures

Objectives

After completing this unit, you will be able to:

- Used control structures besides just if..else statements
- Define various types of repeated execution
- Use break/continue to control individual iterations
- Label statements to control nested iterations

9.1 'switch' Statement

- Selects from multiple choices based on an expression
- The expression must evaluate to an **int** or a type which can be promoted to an **int**
  - **char** type is acceptable
  - "Wrapper" objects are also OK
- The value of the expression determines which case is executed
- The **default** case handles values not matched by other cases

```
switch (expression) {
  case n1:
    // case 1 statements
    break;
  case n2:
    // case 2 statements
    break;
  ...
  case n:
    // case n statements
    break;
  default:
    // default statements
    break;
}
```

'switch' Statement

The values used to identify cases must be unique literal values which can be promoted to an int. The wrapper types Short, Byte, Character, and Integer are also legal.

The expression used in a switch statement may evaluate to a char value and the cases may be identified with char literals such as 'A', 'B', 'Y', 'n', etc.

The statements which execute during the course of a particular case do not have to be enclosed in a separate pair of brackets.

The functionality of a switch statement could also be matched by using an if statement. Using a switch statement is often easier because the expression only needs to be written out once and testing against

each case is implied.

The following prints out what grade someone received:

```
char grade = 'D';
switch (grade) {
    case 'A':
        System.out.println("You received an A");     break;
    case 'B':
        System.out.println("You received a B");    break;
    case 'C':
        System.out.println("You received a C");     break;
    case 'D':
        System.out.println("You received a D");     break;
    case 'F':
        System.out.println("You received an F");     break;
}
```

9.2 Example - switch

```
public static void main(String[] args){
       int x;
       System.out.println(
              "Enter a number between 1 and 3: ");
       x = getInt();
       switch( x ) {
              case 1: System.out.println("One"); break;
              case 2: System.out.println("Two"); break;
              case 3: System.out.println("Three"); break;
              default: System.out.println(
                     "Error: invalid choice of number");
       }
}
```

Example - switch

The default case will be performed if no other cases are selected.

The keyword break will trigger an exit from the switch block.

When a case is finished, the next will start unless there is a break declared.

Without any breaks, all remaining cases will be performed until the end of the switch block, including the default.

The **getInt()** method reads a value from the keyboard.

9.3 Switch "Fall Through"

- Sometimes the "fall through" of a switch statement may be desired

```
// will print the days in order of the gifts, like the song.
// Notice no break statements lets each day "fall through"
for(int day = 1; day <= 12; day++)
{
        switch (day)  {
                case 12: System.out.println("12 Drummers Drumming");
                case 11: System.out.println("11 Pipers Piping");
                case 10: System.out.println("10 Lords a Leaping");
                // other days omitted
                case 2:  System.out.println("2 Turtle Doves");
                case 1:  System.out.println(
                                "1 Partridge in 1 Pear Tree");
                break;
        }
}
```

Switch "Fall Through"

The last case does not need a break statement but it is a good practice to include it so other cases can be added later if needed.

9.4 Using switch "Fall Through" for Multiple Options

- Many times you want the same block of code to execute for a few different options
  - Perhaps letting a user enter upper or lowercase for an option
- You can have some cases "fall through" to others so the same block is executed for multiple options

```
switch (userChoice) {
       case 'q':
       case 'Q':
              System.out.println("Quitting the program");
              timeToQuit = true;
              break;
       case 's':
       case 'S':
```

```
                ...
                break;
}
```

Using switch "Fall Through" for Multiple Options

This avoids unnecessary duplication of code for both cases which should be handled in the same manner.

9.5 Java 7 – Strings in switch Statement

- Prior to Java 7, switch statements could only take some primitive types (and wrapper classes) and enum types for case labels
  - If you wanted to compare to a String you needed to use if/else blocks

```
public int returnNumDays(String month) {
        if ("January".equalsIgnoresCase(month)) {
                return 31;
        } else if ("February".equalsIgnoresCase(month))
{ //...
```

- With Java 7 you can use Strings directly as the labels for cases
  - These will be case sensitive when matching unless you convert the String to compare first
  - This can make for more intuitive code

```
public int returnNumDays(String month) {
        String upperMonth = month.toUpperCase();
        switch (upperMonth) {
                case "JANUARY":     case "MARCH": // other
months
                        return 31;
                case "APRIL":  case "JUNE":  // other months
                        return 30;
                // ...
```

Java 7 – Strings in switch Statement

Besides being easier to write, the compiled byte code is also generally more efficient.

The behavior of matching the cases is the same as calling the String.equals method.

9.6 'for' Loop

- Repeats a set of statements a certain number of times
- The control of the loop is defined within parenthesis after the for keyword
- The initialization, condition, and iteration parts of the control are separated by semi-colons
- More than one statement per control segment can be included separated by commas

```
for (int i = 0; i < 10; i++)
{
        // executed 10 times
        …
}
```

```
for (int i = 0, j = 10; i <
j;
      i++) {
        // executed 10 times
        …
}
```

'for' Loop

The initialization part of the control is executed once before any other parts of the for loop are executed. This segment is often used to declare and initialize a variable which will be used for counting the loop iterations. The condition part of the control is a boolean expression which must be true to execute the body of the for loop. This condition is checked before every iteration of the loop including the first iteration. The iteration part of the control is executed after every successful iteration of the loop. It is executed before the condition expression is evaluated again. The iteration segment is often used to increment or decrement variables.

Sometimes one for loop is nested within another. This is often done with multidimensional arrays. Each loop has a unique index and a complete set of iterations of the inner loop occurs for each iteration of the outer loop:

```
for (int i = 0; i < 10; i++) {
          for (int j = 0; j < 10; j++) {
                    // These statements executed 100 times total
                    …
          }
}
```

An infinite for loop is possible but is generally avoided and must contain a break statement to exit:

```
for (;;) {
          …
          if (timeToBreak) {
                    break;
          }
}
```

9.7 Example - for

```
public static void main(String[] args)
{
	int upperBound;
	int curValue;

	System.out.print("Please enter the number to " +
		"count to: ");
	upperBound = getInt();

	for(curValue=1; curValue <= upperBound; curValue++)
	{
		System.out.print( curValue + " " );
	}
}
```

Example - for

The counter is initialized, tested against the loop condition, and its increment step declared, all within the signature of the for loop. Where the counter is initialized, the variable can also be declared ("int curValue = 1" instead of "curValue = 1") to make a variable with scope only local to the loop.

If the value 9 is entered, what will be the output?

1 2 3 4 5 6 7 8 9

or

1 2 3 4 5 6 7 8

For all of these, the output goes up to 8, since the increment happens at the end of the loop block and then tested before the loop repeats. If the counter reaches the upper bound (9), then it exits the block before printing out any more values.

9.8 'while' Loop

- Repeats a set of statements while a condition is **true**
- The condition expression must return a **boolean** value
- If the condition is initially **false**, the loop block will not be executed at all
- Make sure something changes in the body of the loop to eventually let the condition be false

```
while (expression) {
// These statements repeated
        …
}

double number = 1.2;
while (number < 5.0) {
        number *= 2;
        …
}
```

'while' Loop

The while loop is used to repeat a set of statements when the number of iterations is not as well known as when using a for loop. Unlike a for loop, some variable or condition must change during the course of execution of the loop for the loop to eventually terminate.

An infinite while loop is possible but generally avoided. If declared, a break statement must be used to terminate the loop:

```
while (true) {
        …
        if (timeToBreak) {
                break;
        }
}
```

9.9 Example - while

```
public static void main(String[] args) {
        int upperBound;
        int curValue = 1;

        System.out.print("Count from 1 to what " +
                "integer? ");
        upperBound = getInt();
```

```
        while( curValue < upperBound ) {
               System.out.print( curValue + " " );
               curValue++;
        }
}
```

Example - while

The getInt() method is the same one that reads from the Console In window.

If the integer entered is greater than one, then it will count and display the values from one up to that integer.

If the value 9 is entered, what will be the output?

1 2 3 4 5 6 7 8 9

or

1 2 3 4 5 6 7 8

What happens if the value 1 is entered?

What happens if the value 0 is entered?

9.10 'do…while' Loop

- Similar to **while** loop
- Condition is tested at the end of the loop statement
- Semi-colon at the end of the loop
- Loop statement is always executed at least once

```
do {
     // These statements
     // repeated
     // at least once
     …
} while (expression);

int number = 6;
do {
     // this loop stills
     // executes once
     …
} while (number < 5);
```

'do…while' Loop

One common problem encountered with loops is termed the "fencepost" problem. If you imagine constructing a fence, you always need one more fencepost than sections of fence. For instance, 5 fence sections requires 6 posts and 10 fence sections require 11 posts. Often this means that some part of a process must be performed outside of the loop to ensure the extra "post" is placed. Each type of loop can be used in some way to solve the fencepost problem. The following code uses a do…while loop to print sections and posts of a fence:

```
int count = 0;
int numSections = 5;
do {
          System.out.print("|");       // print post
          System.out.print("-");       // print fence section
          count++;
} while (count < numSections);
System.out.println("|");     // print final post
```

Notice inside the loop that there is a statement to print a post as well as a statement to print a section of fence. Outside the loop there is a single statement to print the final post. This loop always prints out at least one fence section (|-|). Use of a do…while loop makes sense here because other loops might print out only the single final post (|) as a minimum.

9.11 Example - do while

```
public static void main(String[] args)
{
        int upperBound;
        int curValue = 1;

        System.out.print("Please enter the number " +
                "to count to: ");
        upperBound = getInt();

        do {
                System.out.print( curValue + " " );
                curValue++;
        } while( curValue < upperBound );
}
```

Example - do while

This is the same as the previous example, but the while is now a do … while.

If the value 9 is entered, what will be the output?

1 2 3 4 5 6 7 8 9

or

1 2 3 4 5 6 7 8

What will happen if 1 is entered?

If 0 is entered, what will happen?

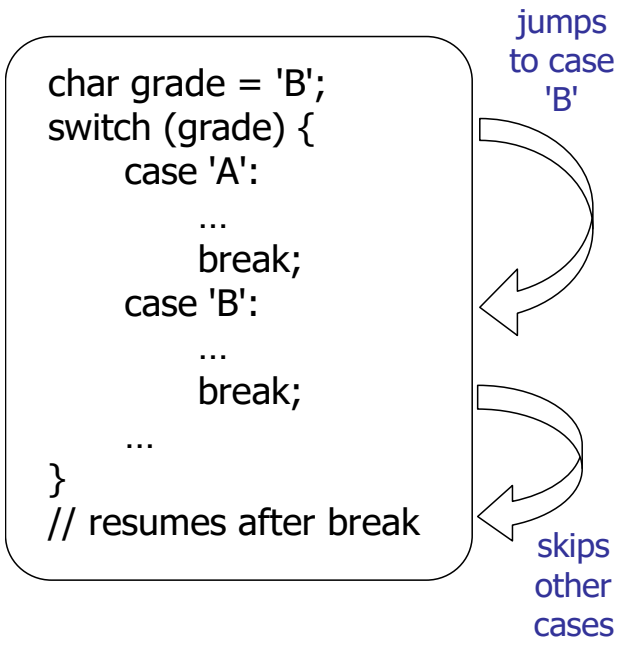
Since the system prints out its curValue first, then tests the condition, 1 will be printed, then exit the loop since one is greater than zero:

1

Of these loop structures, the do…while loop is the only one that performs the block at least once, before deciding to repeat the process.

9.12 Break Statement

- A **break** statement is used extensively with a **switch** statement
- Does what it suggests, "breaks out" of the current **switch** statement
- Moves program execution to the first statement after the **switch** statement
- Can also be used to terminate loops
- Cases of a **switch** statement which do not end with a break "fall through" to the next labeled case



Break Statement

If a break statement is not encountered at the end of a case, execution continues with the statements of

the next case. For this reason, case labels can be thought of as "markers" indicating where a program should "goto" but have no effect on the program afterwords.

9.13 Example - break

```
for( char tmp = 'a' ; tmp < 't' ; tmp++ ) {
       if( tmp == 'k' )
              break;
       System.out.print( tmp + " " );
}
```

- output: a b c d e f g h i j
  - 'k' stops the loop even though the loop condition would have continued if the 'break' were not encountered

Example - break

With a break, the current block of code would be exited.

'p' is incremented to 'q' and triggers the exit.

'q' is not printed since the remainder of the loop is ignored.

9.14 Labeled Statements

- Statements may be labeled with an identifier
- Most often used with control statements to create more complex structure
- Defined by placing identifier and colon before the statement to be labeled

```
OuterLoop:
for (int i = 0; i < 10; i++) {
       // other statements
}
```

- Referred to by other statements

```
break OuterLoop;
```

Labeled Statements

When used without a label, the break statement and a few similar control statements we'll see later only affect the innermost switch statement or loop. To affect something outside of this, they must refer to a pre-defined label. For example, the following code uses an 'OuterLoop' statement label to break out of both nested loops at once:

```
boolean foundIt = false;     // set to true when we wish to exit
OuterLoop:
for (int i = 0; i < 10; i++) {
          for (int j = 0; j < 10; j++) {
                    // do some searching to find if we found something
                    if (foundIt) {
                              System.out.println("We found it!");
                              break OuterLoop;
                    }
          }
}
```

This code would stop both loops once the item was found so the inner block of statements may not be executed for a full 100 iterations.

9.15 Example - Labeled break

```
out:
for( char tmp = 'a' ; tmp < 't' ; tmp++ ) {
        for( int i = 1 ; i <= 3 ; i++ ) {
                if( (i == 2)&&(tmp == 'c' ) )
                        break out;
                System.out.print( tmp + " " );
        }
}
```

- output: a a a b b b c
  - Second 'c' stops both loops

Example - Labeled break

With labels, the way to exit from these blocks of code can be controlled. This way, we can choose when (what conditions apply) and where (in a nested code block) to jump to or exit from.

In this slide, the break sends the control to the out label, and breaks from there. If no label were given, the break would send us to the upper level and continue from there. The output would look like:

a a a b b b c d d d e e e f f f … … r r r s s s

9.16 Continue Statement

- Skips the remaining statements of a loop iteration but does not halt the loop
- May be used with or without a label
- Same use with all types of loops

```
for (int i = 0; i < 10; i++) {
  if (i % 4 == 0) {
    continue;
  }
  // skipped every 4th time
}
```

```
OuterLoop:
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; j++) {
    if (skipTime) {
      // halt inner loop and
      // resume outer loop
      continue OuterLoop;
    }
  }
}
```

Continue Statement

When used with a for loop, the loop control is evaluated just like the loop iteration finished normally. Any statements in the iteration part of the control are executed and the condition is evaluated to determine if another loop iteration should be started.

When used within nested loops, as in the second example above, labeled continue statements halt execution of any inner structures. In the above example, if the variable skipTime were true, the inner loop would be halted no matter what the value of j and the for loop labeled as 'OuterLoop' would skip any statements after the inner loop, increment i, and decide if another iteration is in order.

9.17 Example - continue

```
for( char tmp = 'a' ; tmp < 't' ; tmp++ ) {
       if( tmp == 'k' )
              continue;
       System.out.print( tmp + " " );
}
```

- Output: a b c d e f g h i j l m n o p q r s
  - 'k' is skipped but loop continues

Example - continue

Similar to a previous example, the continue is triggered at ‘k’.

The remainder of the current iteration with ‘k’ is ignored and is not printed, so the next iteration is called.

‘k’ increments to ‘l’ and prints as normal.

9.18 Example - Labeled continue

```
out:
for( char tmp = 'a' ; tmp < 't' ; tmp++ ) {
       for( int i = 1 ; i <= 3 ; i++ ) {
              if( (i == 2)&&(tmp == 'c' ) )
                     continue out;
              System.out.print( tmp + " " );
       }
}
```

- Output: a a a b b b c d d d e e e f . . .
  - Skips the second 'c' but continues the outer loop

Example - Labeled continue

In a previous example, we exit (break) the whole block at the second ‘c’ but here we resume (continue) at the upper level when the second ‘c’ arrives. The tmp character increments, and the counter i is initialized. We essentially ‘skip’ the step of printing the rest of the ‘c’ and continue on to the next letter.

If no label were given the second 'c' would be skipped but the third 'c' would be printed:

a a a b b b c c d d d e e e f f f … … r r r s s s

9.19 Coding Tips - Control Structures

- Use indentation to indicate program structure
- Place opening bracket on same line as statement and align closing bracket with statement
- Always use brackets, even when only a single statement is enclosed
- Always include **default** cases for **switch** statements and use a comment

to indicate when one case "falls through" to the next case

Coding Tips - Control Structures

These tips provide a balance of avoiding excessive white space and providing an easily understandable program.

9.20 Summary

- Loops and other control structures make it easier to write repetitive or complex programs
- Different types of statements exist for different situations
- Statements can be labeled to allow the option of breaking or continuing from within a nested control statement

Chapter 10 - Inheritance

Objectives

After completing this unit, you will be able to:

- Describe inheritance in Java
- Determine which elements of a superclass are inherited by a subclass
- Describe how inheritance makes it possible to "reuse" code
- Describe how all classes are related to the Object class

10.1 Inheritance Is…

- Defining a new class based on an existing class
- A way to provide a generalization of some attributes shared by several classes
- A way to provide concrete relationships between classes
  - Model logical similarities between types of objects
  - "subclass" inherits from "superclass"
  - "derived class" inherits from "base class"

Inheritance Is…

The terms subclass-superclass or derived class-base class are largely interchangeable and depend on the whim of the writer which is used more often.

10.2 Inheritance Examples

- A **BankAccount** might declare shared features while a **CheckingAccount** and **InvestmentAccount** declare unique features not inherited from **BankAccount**
- A **RentalItem** declares common features while **Video, DVD,** and **Game** declare special features

Inheritance Examples

There are many other examples where inheritance would be a way to describe the relationship between types of objects. In general, whenever there are several types of objects which share similar features but also display differences, some type of inheritance will be used to define the relationship.

10.3 Declaring Inheritance

- Inheritance is declared using the extends keyword
  - The subclass "**extends**" the superclass
  - The extends keyword and the name of the superclass are added to the class declaration of the subclass

```
public class CheckingAccount extends BankAccount {
       // rest of subclass definition
}
```

- Java uses single inheritance
  - A class may only "extend" one other class

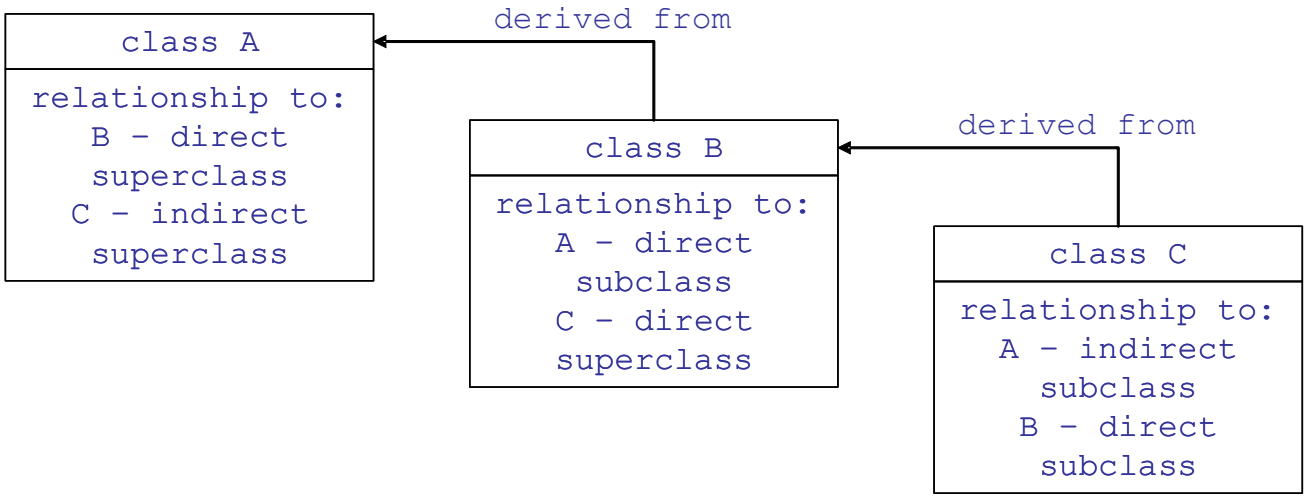
Declaring Inheritance

When defining one class which inherits from another, the subclass only declares those features that are unique from the superclass. Although the subclass only declares these unique features, it should still be thought of as a combination of the unique features it declares and the shared features inherited from the superclass. Exactly what is inherited depends on the access modifier used when defining the element of the superclass, as will be seen in a few slides.

Some object oriented languages allow for multiple inheritance. Java, however, does not and only allows for single inheritance. In other words, one class definition may only extend one other class definition. If more than one class name follows the extends keyword, a compilation error will occur. Other methods for describing common features besides inheritance will be described in the next unit.

10.4 Inheritance Hierarchy

- It is also possible to have multiple levels of inheritance



Inheritance Hierarchy

In the above example, the classes have been declared this way because there is some logical relationship among the three classes which mirrors the inheritance hierarchy. The further down the hierarchy you look, the more specific the definitions become. This is because besides declaring its own features, class B inherits some features from class A. Likewise, class C declares its own features and inherits features from class A AND class B. Exactly what features are inherited and how will be discussed later.

10.5 Access Modifiers Revisited

- The addition of subclasses completes the definition of the access modifiers
  - **public** – accessible to every class in any package
  - **private** – not accessible to other classes
  - **protected** – accessible to every class in same package and subclasses in any package
  - No access attribute – accessible to every class in same package but not accessible to classes in different packages (even subclasses)
- Access modifiers affect fields and methods

Access Modifiers Revisited

Finally, with the addition of packages and inheritance, the definition of the access modifiers is

complete. The definitions of the public and private modifiers are not affected by packages or subclasses but the protected and "default" access modifiers are.

The access modifiers are responsible for what is inherited by a subclass from its superclass as seen in the next slide.

10.6 Inherited Members

- Constructors are never inherited by a subclass
- Fields and methods are inherited according to their access attribute and the package of the subclass
  - Subclasses in the same package inherit fields and methods declared as **public, protected**, or without an access modifier
  - Subclasses in a different package inherit fields and methods declared as **public** or **protected**
- What is inherited determines what is "visible" from within the code of the inheriting class

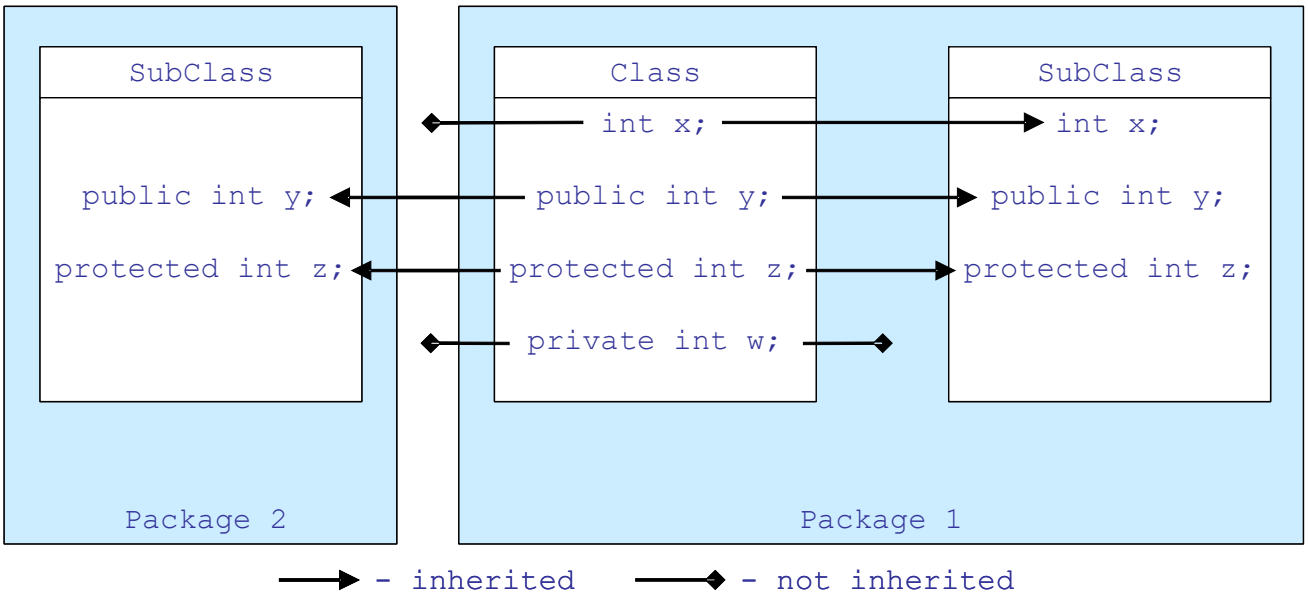
Inherited Members

Only members inherited from a superclass are accessible from within the methods declared within the subclass. Referring to an inaccessible field or calling an inaccessible method from the superclass will result in a compilation error. Inherited members behave as if they were declared in the subclass itself.

Although constructors are not inherited in a subclass, they can be called from within the constructors defined in the subclass. We will see in the next section how to call superclass constructors from the subclass and the benefits this provides.

One way to think about inheritance is an "automatic copy/paste" of code into the inheriting class. The access modifier of methods and fields determine what is copied into the subclasses.

10.7 Inherited Members



Inherited Members

Although only fields are shown above, the same behavior applies to methods.

10.8 Instances Of A Subclass

- Although a subclass inherits only certain members from a superclass, instances of the subclass type contain all members declared in the superclass and subclass
  - Inheritance affects the accessibility of members from the superclass rather than what members exist in instances of a subclass type
- Even inaccessible members are still vital to the state of the subclass instance
- Often inaccessible members are accessed from other methods accessible within the subclass
  - A public **getXXX**() method to access a private field

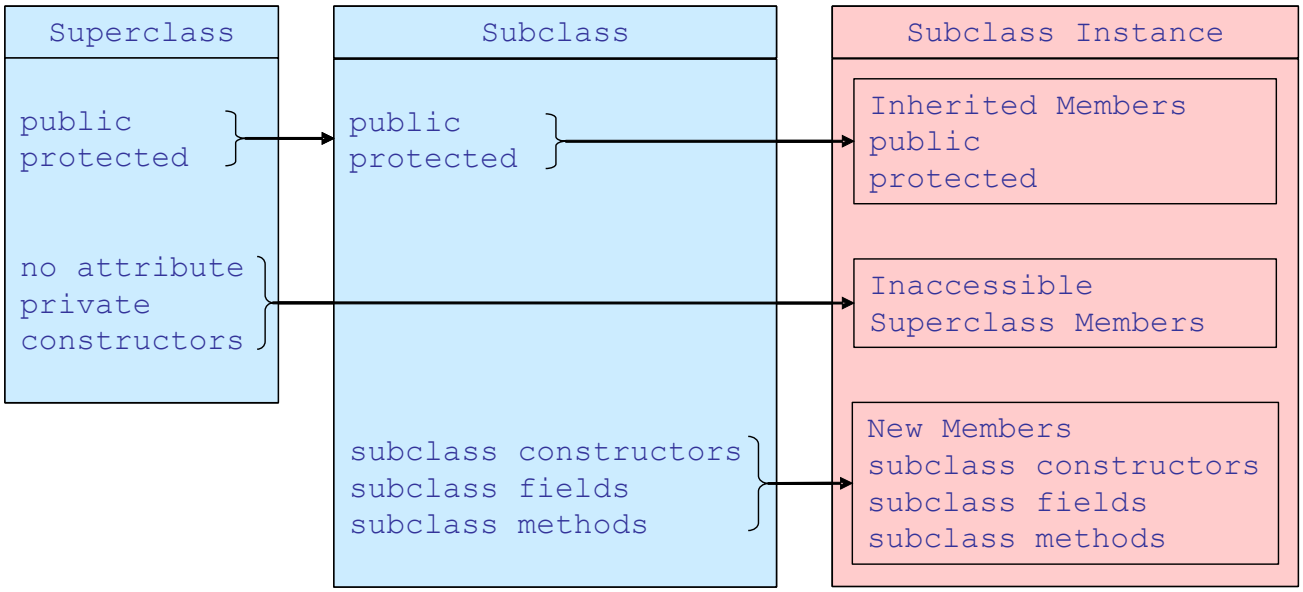
Instances Of A Subclass

One way to think about the behavior described above is to think in terms of how inheritance is

declared. A subclass "extends" the superclass by adding to the definition of the superclass. If part of the superclass definition were removed by removing superclass members that were inaccessible in the subclass, the subclass would not be truly "extending" the superclass.

Going back to the bank accounts example, the BankAcnt class could declare a field to store the bank account number. This field should be declared private and therefore not inherited in a CheckingAcnt subclass. If an instance of the CheckingAcnt type did not contain a field for storing the account number, the benefit of extending the BankAcnt class would be lost. The field is still included in the state of the CheckingAcnt instance but must be accessed in a different way (by calling getAccountNum() for instance).

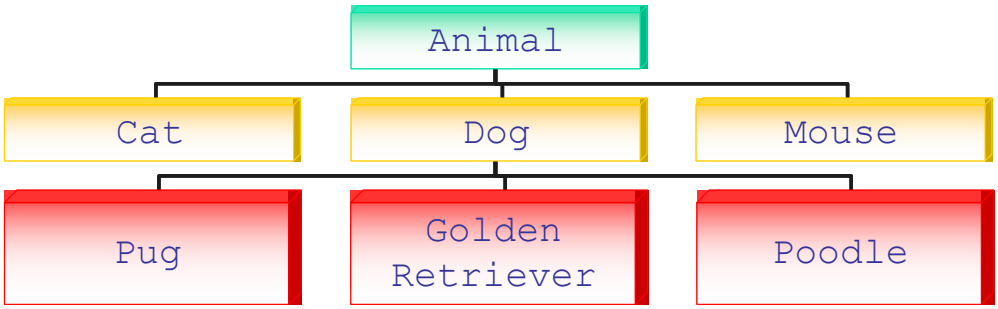
10.9 Instances Of A Subclass



Instances Of A Subclass

In the diagram above, members listed with no access attribute are added to the 'Inaccessible Superclass Members' group. This is true if the subclass is in a different package than the superclass. If they were in the same package, members with no access attribute would be in the 'Inherited Members' group.

10.10 Example Of Inheritance



Example Of Inheritance

The diagram above demonstrates the relationships that can be created with inheritance. Although there are things that make cats, dogs, and mice different, they all have similarities that make them animals. Although Pugs, Golden Retrievers, and Poodles are all dogs, they all have unique features that make them different from each other.

10.11 Role In Reuse

- Inheritance provides more efficient programs by allowing common code to be described in superclasses and differences to be coded in subclasses
- Specialized definitions can be written based on generalized definitions
- One of the primary mechanisms for writing "less code, better programs"

10.12 Overriding Methods

- Sometimes a subclass may want to change the behavior of a method defined in the superclass
  - It can do this by "overriding" the method
- To override a method, a subclass defines a method of the same signature (name, number and type of parameters)
  - This provides an alternate definition for that particular class of objects

```
public class Superclass {
```

```
        public void overrideMe() {...}
}

public class Subclass extends Superclass {
        // changes the behavior for this subclass
        public void overrideMe() {...}
}
```

10.13 @Override Annotation

- If the method signature of a method does not match a method defined in superclasses you would not be overriding a method.
  - This can create a tough to debug problem where you think the method in the subclass is being called but Java is calling the superclass method
- You can add the '@Override' annotation to a method declaration so the compiler checks that the method does indeed override a superclass method.
  - If the method does not have a match in a superclass that it overrides a compilation error is reported.

```
public class Superclass {
        public void overrideMe() {...}
}

public class Subclass extends Superclass {
        // This method would not compile
        @Override
        public void overrideMe(String extraParam) {...}
}
```

- The '@Override' annotation is not required to actually override the method, it just has the compiler check that the method being defined does override something from a superclass

@Override Annotation

This annotation was added in Java 5 when annotations were first introduced.

Although Java development tools can help create the signature of the method to override adding the annotation will ensure that if the method is modified in the future it is always overriding a superclass

method.

It is always better to have code that won't compile compared to code that compiles but produces unexpected behavior when run.

10.14 The super Keyword

- Allows a subclass to refer to its superclass
- To access a member variable of the superclass:

```
super.nameOfVariable;
```

- To access methods of the superclass:

```
super.nameOfMethod(<parameters>);
```

- To call constructors of the superclass:

```
super(<parameters>);
```

The super Keyword

Calling a constructor of the superclass is only allowed in a constructor of the subclass and must be the first method called.

Accessing the member variables or methods of the superclass is only possible if the access modifiers allow it.

10.15 Example - super Keyword

```
public class Account {
  public Account(String name, int accountNumber,
       double balance) {
    // set fields to values of constructor parameters
  }
  // other methods omitted
}

public class SavingsAccount extends Account {
  public SavingsAccount(String name, int accountNumber,
```

```
      double balance, double interestRate) {
    super(name, accountNumber, balance);
    this.interestRate = interestRate;
  }
}
```

10.16 Problems with Constructors

- If you do not explicitly call a superclass constructor as the first line of a subclass constructor, the compiler will automatically do this for you
  - It inserts a call to the **super()** constructor
- The problem is this constructor may not exist in the superclass!
  - This constructor will not exist if you have defined your own constructors but not this one
  - This will cause compilation problems

10.17 Problems with Constructors

```
public class Account {
  public Account(String name, int accountNumber, double
balance) {
    // set fields to values of constructor parameters
  }
    // no constructor with zero arguments
}

public class SavingsAccount extends Account {
  public SavingsAccount(String name, int accountNumber,
double balance, double interestRate) {
    // automatically adds super();   COMPILE ERROR!!
    this.interestRate = interestRate;
  }
}
```

Problems with Constructors

This code is the same as the last example except the constructor of the SavingsAccount class does not explicitly call a constructor of the superclass. The compiler inserts a call to super() and the compiler will not be able to compile the class.

10.18 Limiting Subclasses

- The **final** keyword can be used
  - final classes can't be subclassed
  - final methods can't be overridden
- You can use the **final** keyword to prevent other code definitions from redefining things that should not be modified
  - Anything that extends a class will inherit anything declared as final exactly as declared in the superclass

10.19 Calling Methods in Constructors

- It is best to avoid calling any methods in constructors
  - This is true because of inheritance
- If a subclass extends another class and overrides one or more methods the behavior may be changed
  - This modified behavior may access data that is not yet initialized when called as part of the superclass constructor
- You could add the 'final' keyword to the methods to prevent overriding but this is usually only for specific situations and should not be a default practice

Calling Methods in Constructors

The code below would produce unintended behavior:

```
public class Account {
          public Account(double initBalance) {
                    // Might call the subclass version
                    setBalance(initBalance);
          }
```

```
          public void setBalance(double newBalance) {
                    balance = newBalance;
          }
          private double balance;
}

public class MinimumBalanceAccount extends Account {
          private double minimum = 1000.0;
          public MinimumBalanceAccount(double initBalance, double minimum) {
                    super(initBalance);
                    this.minimum = minimum;
          }
          public void setBalance(double newBalance) {
                    if (newBalance < minimum) {
                              System.out.println("You can't do that");
                    } else {
                              balance = newBalance;
                    }
          }
}
```

Calling the subclass constructor may call the subclass version of the 'setBalance' method as part of the call to the superclass constructor. The problem is the value for 'minimum' has not yet been initialized using the value passed in to the subclass constructor. The following code would not behave as expected:

```
MinimumBalanceAccount act = new MinimumBalanceAccount(750.0, 500.0);
```

The sample code above uses a 'set' method to show the problem but it might be the same with calling any method from a constructor.

10.20 The Object Class

- In Java, the **Object** class is the common "ancestor" for all other classes
  - If a class does not extend another class it automatically extends the Object class
- There are seven public methods and two protected methods that are inherited from the **Object** class
  - public – **toString(), equals(), getClass(), hashCode(), notify(), notifyAll(), wait()**
  - protected – **clone(), finalize()**
  - These methods are present in every Java class

The Object Class

When a class does not explicitly extend another class an implicit 'extends Object' is understood by the compiler.

The methods above that are inherited from the Object class are described in full detail in the API documentation for the Object class. Most deal with multi-threaded behavior or are used by the run-time Java environment.

The most useful method inherited from the Object class is the toString() method. This method returns a String representation of the instance it is called on.

10.21 Summary

- Inheritance is the means by which one class "extends" another
- A subclass extends a superclass
- Inheritance is declared by adding the keyword extends and the name of the superclass
- Inheritance can be several layers deep
- Access modifiers control which elements of a superclass are inherited by a subclass
- Even if elements of a superclass are not inherited, they are still present in instances of the subclass type
- All classes are subclasses of the Object class

Chapter 11 - Arrays

Objectives

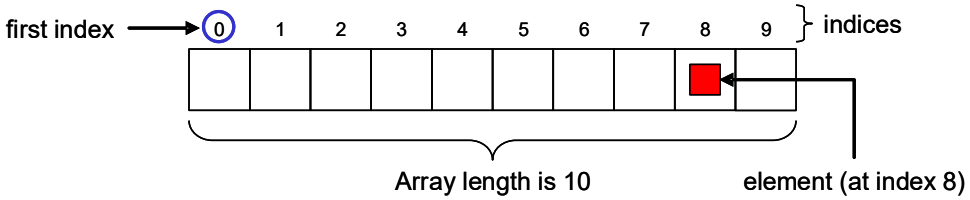
After completing this unit, you will be able to:

- Define and initialize arrays
- Use multidimensional arrays
- Use arrays in loops with the for-each loop

11.1 Arrays

- Named set of variables of same type
- Size fixed when created
- Integer indexed from zero

```
int[] sampleArray = new int[10];
sampleArray[8] = 36;
```



Arrays

Arrays can be declared for primitive types or objects. No matter what type the array is declared as, every value, or array element, is of the same type. Just like single-value variables, arrays must be declared with the array type and an identifier before being used.

To refer to an element of the array, follow the array name with the index of the element enclosed in square brackets. The index value used may be an expression but must be zero or a positive int value. There may also be spaces between the array name and the square brackets:

System.out.println(sampleArray[3]);

System.out.println(sampleArray [3]);

Array elements may be used in expressions and operations in the same manner as simple variables of the same type.

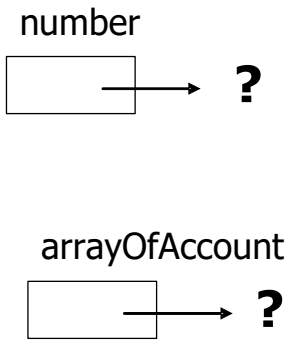
There are actually several ways to declare arrays. The position of the square brackets indicating an array can be placed in different locations and can be preceded by spaces. All of the following array declarations are legal and will not generate compiler errors:

```
int[] firstArray;
int [] secondArray;
int thirdArray[];
int fourthArray [];
```

Although all of the above examples are legal arrays, the first example is typically preferred because it most clearly indicates the type of the variable as an array of int values.

11.2 Declaring Arrays

- Declare arrays of primitive or class data types
- Java arrays are declared with square brackets
  - Can be after variable name or after data type




```
int number[];
Account [] arrayOfAccount;
```

- Creates space for a reference

Declaring Arrays

Java does not allow you to specify the size of an array when declaring an array variable. You must explicitly set the size of the array with a new operator or by assigning a list of items to the array at the time of creation. This will be discussed next.

Take note that in array declaration, you can place the square brackets after either the variable type or identifier.

The declaration of an array creates a reference that can be used to refer to an array.

11.3 Populating Arrays

- Two ways to populate arrays:
  - Declare size with 'new' keyword, initialize each value

```
int[] sampleArray = new int[5];
sampleArray[0] = 36;
sampleArray[1] = -24;
...
```

- Initialize with list of literals (size taken from list)

```
int[] sampleArray = {36, -24, 135, -335, 12};
```

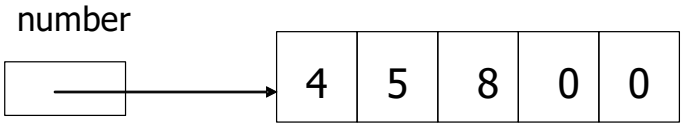
Populating Arrays

When created, all elements of an array are automatically initialized with default values. The default value for numeric types is zero, for a boolean array it is false, and for an array of char values it is the empty character '' or '\u0000'. If an array is of an object type, each element is initialized with a null reference so each element must be explicitly initialized to contain a valid object.

The second option to create and initialize an array is termed an "array initializer".

11.4 Accessing Arrays

- Use the array name and a subscript.
- The subscript is zero based.




```
number[0] = 4;
number[1] = 5;
number[2] = 8;
```

Accessing Arrays

In accessing elements in an array, the syntax is:

```
arrayName [subscript]
```

11.5 Arrays of Objects

- If an array is declared of a class each element can store a reference to that type of object
- The value of each element will start as 'null' unless it is initialized with a real object
  - This can be done with a constructor or from an object returned from a method just like non-array variables
- Methods can be called after indicating an array element

```
Account accounts[] = new Account[3];
accounts[0] = new Account(1000);
accounts[1] = lookupAccount(14);
// accounts[2] is still null

accounts[0].deposit(300);
```

11.6 Array Length

- The length of an array is a property of the array which is obtained by placing '.length' after the array name

```
sampleArray.length
```

- Often used within the condition of a loop

```
for (int i = 0; i < sampleArray.length; i++) {…}
```

- The length of the array is not a legal index and will cause an Exception

```
sampleArray[sampleArray.length]   // illegal index
```

Array Length

Obtaining the size of an array by using the length property is very desirable because it allows a program to adapt to an array of various lengths dynamically. Using an explicit value instead of the length property introduces possible errors if the program is modified in the future.

Because the last legal index of an array is one less than the length, the '<' operator is the most common way to test if an index is legal. The '<=' operator may also be used but only if compared against a value

one less than the array length. Because this is a more complex expression, the '<=' operator is typically not used.

Because arrays are indexed from zero, it is common to have incorrect code which is "off by one". This is most common when some type of expression is used to determine the index to use to access an array element. Every array index expression should be examined to determine if the correct elements of the array will be accessed.

One thing that you have to take note of is that Java will have a runtime error if you will try to access an array beyond its size. This is commonly known as out of bounds error.

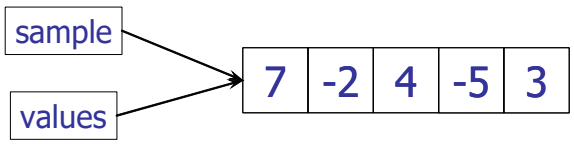
11.7 Coding Tips - Arrays

- When declaring arrays, place square brackets immediately after type of array
- Avoid extra spaces before square brackets
- Declare array size dynamically if possible and only immediately before initializing the array
- Make use of length property whenever possible
- Check to be sure the array exists before attempting to access properties or elements

11.8 Array References

- Array variables refer to the memory location used to store the values
- Two array variables can refer to the same array

```
int[] sample = new int[5];
int[] values = sample; // copies the reference to the array
```



Array References

In the code above, the new keyword in the first line is what allocates the memory needed for the array of five int values. The second line simply copies the reference to the array and does not allocate

another array. Referencing elements of the array using the sample or values identifiers would be equivalent.

Although array identifiers only hold references to the memory location of the array, this location is not available for direct manipulation as in some other programming languages such as C. The array must be manipulated by referring to individual elements of the array or creating another new array for the identifier to refer to. If creating another array, even of a different size, the same identifier may be reused:

```
// 1st new array created
int[] reuseMe = new int[5];

// 2nd new array of different size
reuseMe = {3, -24, 45, 153, 234, -25, 29};

// 3rd new array created
reuseMe = new int[100];
```

11.9 Multidimensional Arrays

- Used to record multiple sets of related information. For example, five measurements over a period of 50 days

```
double[][] measurements = new double[5][50];
```

- Useful when combined with loops

```
int[][] multiply = new int[15][15];
for (int i = 0; i < 15; i++) {
       for (int j = 0; j < 15; j++) {
              multiply[i][j] = i * j;
       }
}
```

Multidimensional Arrays

Determining the length of a multidimensional array is a little more difficult. To obtain the length in the first dimension, the standard notation is used:

```
multiply.length
```

To obtain the length in the second dimension, the .length property must refer to a specific element of the array in the first dimension. This is done by adding a single pair of square brackets to the identifier and using a legal array index for the first dimension:

```
multiply[4].length
```

To follow the practice of always using the array length as the condition in the loop instead of an explicit value, the second code example from the slide should be written as:

```
int[][] multiply = new int[15][15];
for (int i = 0; i < multiply.length; i++) {     // length of first dimension
          for (int j = 0; j < multiply[i].length; j++) { // length of second
dimension
                    multiply[i][j] = i * j;
          }
}
```

Arrays of more than two dimensions are also possible by adding additional sets of square brackets. For example, a four dimensional array would be:

```
boolean[][][][] answerMatrix = new boolean[5][15][25][10];
int threeDSize = answerMatrix[0][0].length;     // assigned a value of 25
```

11.10 Arrays Of Arrays

- Multidimensional arrays are actually "arrays of arrays"
- Each array element of the first dimension is another array
- These secondary arrays do not need to be of the same length
- Not all array dimensions need to be allocated at the same time
- This allows for arrays of various "shapes". For example, this code creates a triangle array:

```
int[][] triangle = new int[5][];  // allocates only 1st dimension
// allocate each element in 1st dimension with new array
triangle[0] = new int[1];       // a 1 element array
triangle[1] = new int[2];       // a 2 element array
triangle[2] = new int[3];       // a 3 element array
triangle[3] = new int[4];       // a 4 element array
```

Arrays Of Arrays

Although all dimensions of an array do not need to be allocated at the same time, they do need to be allocated in a certain order. Arrays in the lower dimensions need to be allocated before creating new arrays for any higher dimensions. Using the "array of arrays" notion, you can imagine that you can't create an array for a higher dimension if you do not yet have a location to place it in. For example, the following code would generate a compiler error:

```
int[][] multi;
```

```
multi[0] = new int[5];
   // error, 1st dimension isn't allocated
```

11.11 Copying Arrays

- Use the arraycopy method of the java.lang.System class to efficiently copy data from one array to another
  - public static arraycopy (Object source, int srcIndex, Object dest, int destIndex, int length)

Copying Arrays

The two Object arguments indicate the array to copy from (source) and the array to copy to (dest). The three integer arguments indicate the starting location in the source (srcIndex) and the destination array (destIndex), and the number of elements to copy.

11.12 For-Each loop

- Java 5 added a for-each loop that provides an easy way to iterate over an array
- A temporary variable that will be available in the loop body is declared before the ':'
- The array to iterate over is declared after the ':'

```
int[] values = new int[10];
for (int val : values) {
       // Execute for each item
       System.out.println(val);
}
```

For-Each loop

Java automatically iterates over each item of the array and assigns the value to the 'val' variable. For each item, the body is executed.

One situation where you would not be able to use the for-each loop is if you need to access the current index in the array within the body of the loop. This may be required if you are modifying the values stored in the array or performing a calculation which includes the index of the array. If you just need to

perform a basic action for each value in an array the for-each loop is very elegant.

11.13 Command Line Arguments

- When a standalone Java program (with a 'main' method) is run, additional arguments to the command can be provided
  - These additional command line arguments are passed to the String[] parameter in the main method
  - The command line arguments can be used in the code to dynamically change behavior of the code based on the values passed in
- The following code:

```
public class CopyFileContents {
   public static void main(String[] args) {
      if(args.length != 2) throw
          new Exception("Need 2 file names as arguments");
      String fileToRead = args[0];
      String fileToWrite = args[1];
      // Use the file names passed from the command line
```

- Could be called like this to dynamically supply the file names to act on:

```
java CopyFileContents ReadFromThisFile.txt output.txt
```

Command Line Arguments

The main method does not need to even look at the array passed as a parameter but it can be used if needed.

11.14 Variable Arguments

- One problem in overloading methods is when the number of parameters varies
  - This may be common when you are performing functions on lists of items that vary in length
- In the past the only way to reasonably do this was to pass an array as a method parameter
  - Sometimes this is overhead if you need to create the array just to pass

as a parameter

- Java 5 added the notion of variable arguments to avoid this
  - You can add an ellipsis (...) between the type and name of the parameter
  - Java automatically creates an array for the parameter
  - Only the last parameter can be declared this way

Variable Arguments

The benefit of variable arguments is most apparent in the code that calls the method. You can pass an array for the parameter if it already exists but you do not need to construct one just to pass the values.

11.15 Variable Arguments Example

```
public int sum(int ... args) {
       int total = 0;
       for(int value : args) {
              total += value;
       }
       return total;
}

System.out.println(sum(12, 14, 83));
System.out.println(sum(new int[] {12, 14, 83}));
int[] test = {12, 14, 83};
System.out.println(sum(test));
```

Variable Arguments Example

All of the calls to the sum method above would behave exactly the same way. The first call to the sum method is much easier since we do not need to construct an array for the list of values. If an array already exists, as in the last example, it can also be passed in as the parameter.

11.16 Summary

- Arrays provide a way to store multiple values of the same type

- The 'new' keyword along with a size can be used to create the space for an array
- Arrays are accessed using the array index
- The 'length' property of an array can make code which uses arrays generic and able to adapt to arrays of different size
- The for-each loop can help iterate over arrays

Chapter 12 - Commonly Overridden Methods

Objectives

After completing this unit, you will be able to:

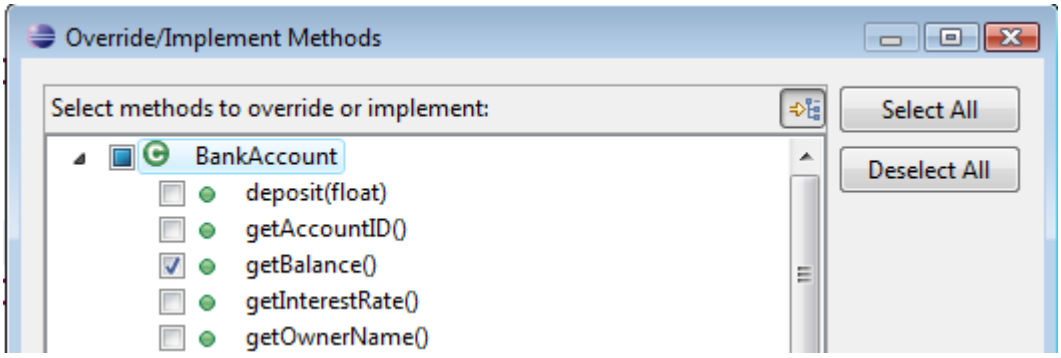
- Describe some of the common methods from the Object class that may be overridden
- Provide implementations in your own class that behave more appropriately than the default implementations

12.1 Overriding Methods

- In Java it is common to change the behavior of a method by "overriding" it
  - This provides an alternate definition for that particular class of objects
- There are some common methods that are frequently overriden
  - toString()
  - equals(Object o)
  - hashCode()

12.2 Using Eclipse to Override Methods

- The Java editor in Eclipse can automatically add a new method stub that will override a method in a superclass
  - All you would need to do is provide the method implementation
- You can select '**Source** → **Override/Implement Methods...**' to use this tool
  - Check the methods you want to override



12.3 toString()

- The **toString()** method is called on an object whenever a String representation is needed
  - This is common when printing information out

```
System.out.println(account.toString());
```

  - Java can also call this method automatically

```
System.out.println(account);
```

toString()

The two statements above are equivalent. Java automatically substitutes a call to toString when it needs to represent the account object as a String when printing it out. Often the second statement is easier to write.

12.4 toString() in Object

- The Object class defines the **toString()** method
  - This means it is defined for every class since every class inherits in some way from Object
- The version in the Object class is not very useful as it just prints out the type of object and memory address
  - To provide a better String representation you must override this method

```
com.simple.account.BankAccount@63232323
```

toString() in Object

Even though every object would have this method it is not useful for display unless you provide an alternative implementation.

12.5 Overriding toString()

- To provide your own definition of the **toString** method you must provide a method with the signature below

```
public String toString()
```

- You can combine values of instance variables with any other characters you want to form one String

```
@Override
public String toString() {
   return "Account #" + getID() + ", Balance: $"
   + getBalance();
}
```

Overriding toString()

You can also provide special characters like the '\n' newline character but remember that the String you return will probably be used as is. Often it is easiest to return the simplest String possible without any additional formatting.

12.6 Comparing Objects

- To compare objects, the '==' operator does not function as expected
  - This is because for objects, the '==' operator checks to see if the variables refer to the same object or different objects
  - The '==' operator does not compare the state of two different objects
- To check if two objects contain the same state and can be considered "equal" the **equals(Object o)** method is used

```
Integer first = new Integer(5);
```

```
Integer second = new Integer(5);

first == second  → false
first.equals(second);  → true
```

Comparing Objects

Notice the manner in which the equals method is used. The method must be called by first referring to a specific Java object, in this case the String object first. After the object reference is indicated, the '.' operator is added along with the name of the method being called on the object. The set of parenthesis indicates this is a method being called and contains any parameters used in executing the method, in this case the String object second which is being tested for equality.

For Strings, the equals method is case-sensitive and will return a false value even if the only difference between two Strings is the case of one character. If you wish to compare Strings without regard to case you can use the equalsIgnoreCase(String) method:

```
String lowerCase = "test case";
String upperCase = "TEST CASE";
boolean withCase = lowerCase.equals(upperCase);  // initialized to false
boolean ignoreCase = lowerCase.equalsIgnoreCase(upperCase); //  initialized to true
```

12.7 Using == vs. equals(..)

- The == operator returns true if both operands refer to the same object instance (identity)
- The Object version of the **equals(..)** method behaves exactly like the == operator
- The **equals(..)** method is overridden in classes for comparison of the values of two different objects (equality)

Using == vs. equals(..)

The default equals() method inherited from the Object class compares the memory location (references) of the objects you are comparing. If you want to compare the internal values of the objects as the criteria for being equal, you must override this method with your own.

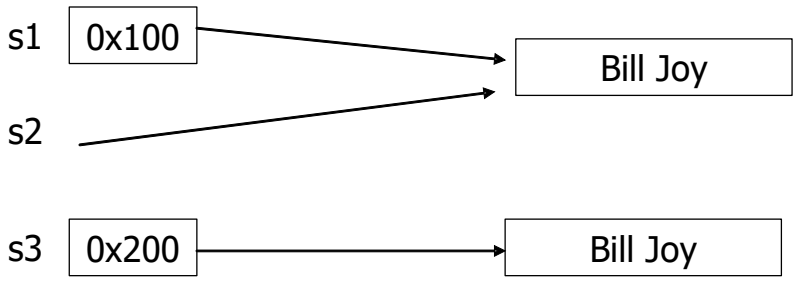
12.8 Using == vs. equals(..)

```
Account s1 = new Account("Bill Joy");
Account s2 = s1;
```

```
Account s3 = new Account("Bill Joy");

s1 == s2;            // returns true
s1 == s3;            // returns false

s1.equals(s2);       // returns true
s1.equals(s3);       // returns true
```



Using == vs. equals(..)

In the above given example:

The == operator checks for equivalent comparison; that is, they refer to the same object.

The equals() method checks for content comparison. The behavior below assumes that the Account class has overridden the equals method to behave differently than the implementation provided by the Object class.

12.9 Overriding equals(..)

- To override the **equals(..)** method you provide a method with the signature below

```
@Override
public boolean equals(Object o)
```

- Since the parameter is of type Object you must also use the **instanceof** operator to check what type of object was passed in
  - If an object is not an 'instanceof' the same class they generally would not be considered equal

```
if(!(o instanceof Account)) return false;
```

- You also want to check that the object passed in is not null

12.10 Complex Comparisons

- It is generally easiest to say two things are not equal
  - All you need is one difference to state this
- For classes that have lots of state to compare check each element separately
  - If anything is found to be different, immediately return **false**
  - Return **true** at the end of the method to indicate all things checked were the same
- For classes that have other objects as instance variables, call the equals method on the other classes to make your method cleaner

```
if(!getName().equals(other.getName()))
        return false;
```

Complex Comparisons

The last statement on the slide means that the example is "cleaner" because you are using the implementation of the equals method provided by the other class, in this case the String class.

12.11 equals(..) Example

```
// Other Account definitions omitted
@Override
public boolean equals(Object other) {
        if(other == null) return false;
        if(other == this) return true;
        if(!(other instanceof Account)) return false;
        // Cast for convenience
        Account otherAct = (Account)other;
        if(!getFirstName().equals(otherAct.
                getFirstName())) return false;
        if(getBalance() != otherAct.getBalance())
```

```
            return false;
        if(getID() != otherAct.getID()) return false;
        return true;
}
```

equals(..) Example

If your class is a subclass of another class you might want to consider a call to super.equals(..) as part of the implementation in the subclass. Only do this if you know there is a reasonable implementation of the equals method in the superclass.

Generally it is advisable to name the parameter something that will make it clearer that you are comparing state from the other object. Using the parameter 'other' above will make any calls to get state of that object similar to:

```
other.getFirstName()
```

Once you have checked that the object passed as a parameter is an instance of the correct class it is good to create another variable that casts it to that type. This will make the rest of the code in the method much easier.

12.12 hashCode()

- The **hashCode()** method returns an int

```
@Override
public int hashCode()
```

- This value is used when organizing groups of objects using a method called "hashing"
  - This usually involves splitting up or sorting a group of objects using the value returned from this method
  - A pharmacy putting prescriptions ready for pickup in buckets by the first letter of the patient last name is an example of hashing
- If two objects are equal according to the **equals(..)** method, calling **hashCode()** on the two objects must return the same value
  - If different hash codes are returned, the process which is organizing the objects with "hashing" will assume the objects are different

```
a.equals(b)  implies  a.hashCode() == b.hashCode()
```

- The reverse is not true, if the hash code of two objects is the same it does not mean that the two objects would be .equals(..)

```
a.hashCode() == b.hashCode()  does not imply  a.equals(b)
```

12.13 Overriding hashCode()

- The default behavior of the **hashCode()** method is also often based on the address in memory of the object instance
- If you are overriding the **equals(..)** method this default implementation will not meet the requirements on the previous slide
  - This means you must override **hashCode()** also
- In your implementation try to return a range of various values
  - This will make any "hashing" operation more efficient

Overriding hashCode()

If your hashCode implementation only returns a few values any "hashing" operation will not be very efficient as it will not be able to split large groups of objects into more manageable sub-groups very well.

The following page has a good algorithm for calculating hash code values:

`www.linuxtopia.org/online_books/programming_books/thinking_in_java/TIJ313_029.htm`

12.14 hashCode() Example

```
public class Video {
   // other methods/constructors omitted

   private String title;
   private String category;

   public int hashCode() {
      // use prime numbers and hash code of properties
      int result = 17;
      result = 37 * result + getTitle().hashCode();
      result = 37 * result + getCategory().hashCode();
```

```
        return result;
    }
}
```

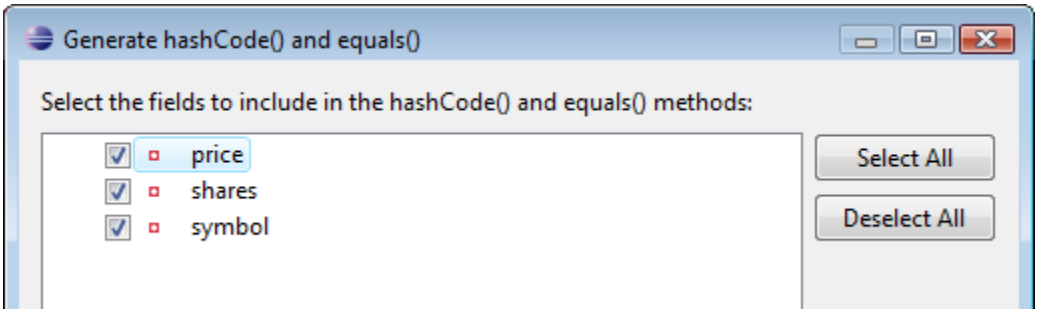
hashCode() Example

The value returned as a "hash code" does not have to be unique. It is not the same as a unique identifier. The ID is used in the example above simply because it is already an int and convenient.

The most important thing to remember about what to return as the hashCode value is to use something that will vary quite a bit over all of the possible instances. If our Video example had a 'price' this would probably not be good to use since it may not vary much.

12.15 Generating equals and hashCode

- Eclipse can automatically generate the equals(..) and hashCode() methods
  - You only need to select the fields that should be compared
  - Eclipse can apply a complex algorithm to generate proper behavior without the risk of an incorrect implementation
- Select **Source → Generate hashCode() and equals()** and then select the fields to compare



- If you already have a equals and hashCode method Eclipse will confirm that you want to regenerate them
  - This is best to do if you have changed the fields of the class

12.16 Summary

- Overriding methods changes the behavior to make sense for a subclass

- toString(), equals(..), and hashCode() are common methods to override

Chapter 13 - Exceptions

Objectives

After completing this unit, you will be able to:

- Use exceptions to write more robust code
- "Throw" exceptions to indicate problems
- "Catch" exceptions for problems that may occur
- Declare your own exception classes for more application-specific problems

13.1 What is an Exception

- Exceptions are the way Java notifies the program that something has gone wrong
- Used to catch things that go wrong that can't be detected at compile time
- Exceptions are categorized so that we can narrow down what went wrong
- They give us an opportunity to solve the problem rather than crash

What is an Exception

Exceptions simplify our code by allowing us to present the most frequent code path in a simple and readable way, then put other code paths .

For instance, if we open a file, then write to it, the code should look like:

```
OutputStream os=new FileOutputStream("destination");
PrintWriter pw=new PrintWriter(os);
pw.println("Hi there!");
```

But what happens if we can't open the file (maybe the directory doesn't exist)?

In 'C' it might look something like:

```
FILE *fp=fopen("destination");
if (fp==null)
```

If there is a call for one value to divide another, division by zero is a possibility. When this happens, an exception is thrown; a message alerting the application or the user that something went wrong.

13.2 Benefits

- Avoids the need to check for status code returned by every method
- Separates error handling code from the normal code
- Allows a clean path of error propagation
- Makes your programs more robust. Unlike error codes, exceptions cannot be implicitly ignored
- Systematic approach to error handling

Benefits

A return code strategy has weaknesses because the code that calls the method would have to know what the return code means and there is nothing that would prevent code calling a method from ignoring the return code. Java methods also often return objects as the result of the method and could not return an object and a return code.

With Exceptions Java methods can declare "I will give you back this kind of data/object unless this type of error occurs".

Exceptions assist in making your program more readable, compared to traditional strategies of testing for conditions before executing an instruction.

13.3 The java.lang.Exception Class

- Contains four constructors, namely:
  - Exception()
  - Exception(String message)
  - Exception(Throwable rootCause)
  - Exception(String message, Throwable rootCause)
- Methods to enable you to access messages and the stack trace:
  - getMessage()
  - printStackTrace()
  - toString()

The java.lang.Exception Class

Exception(String message) - By passing a message through the second constructor, the exception handler can print the message to the screen or log, allowing more meaningful tracking of errors and their causes.

Given this as a template, you can create your own custom exceptions for handling certain situations.

13.4 How to Work With Exceptions

- A program raises an exception using the throw statement:

```
throw <exception object>;
```

- The exception object must be of a class derived from java.lang.Throwable
- To be notified of an exception, the program should use the try/catch block

13.5 Example Exception Handling

```
int x = readInt();
int y = readInt();

try {
        if (x >= y) {
                throw new Exception("x too large");
        }
        // only get here if x < y
        System.out.println("Your first value: " + x +
                " was less than the second: " + y);
} catch(Exception e ) {
        System.out.println(e.getMessage());
}
```

Example Exception Handling

An exception is raised if x is greater than or equal to y. The portion of code within the catch block is executed if and when the exception is thrown.

13.6 The try-catch-finally Statement

- The try block contains code that may throw an exception. This is the 'usual' code path
- If an exception is thrown, execution leaves the 'usual' path and jumps to a 'catch' block
- A catch block handles the exception if the type of its parameter is the class of the exception or a superclass of the type of the exception
- The finally block is always executed no matter how the program control exits the try block

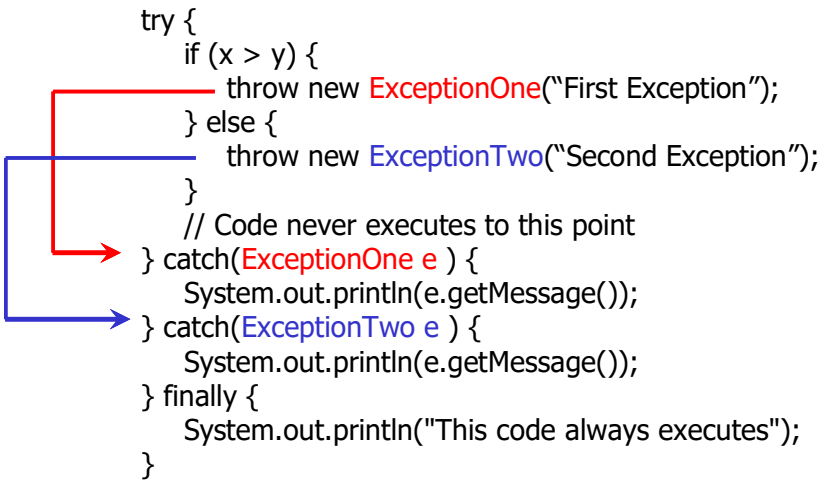
The try-catch-finally Statement

The **try block** represents the execution when everything goes as planned. It should be simple and readable.

When a method throws an exception, the **catch block**(s) hold the response to the exception. The virtual machine runs the first catch block whose variable declaration is compatible with the exception that was thrown in the try block.

The **finally block** is always executed, whether there was an exception thrown or not. It is often used to close resources used in the try block, so as to avoid having to duplicate code both in the try block and the catch blocks.

13.7 Flow of Program Control



Flow of Program Control

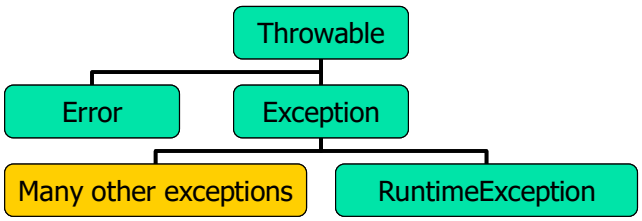
If an exception of type **ExceptionOne** is thrown the flow of control transfers to the **catch(ExceptionOne e)** block.

If an exception of type **ExceptionTwo** is thrown the flow of control transfers to the **catch(ExceptionTwo e)** block.

The **finally** block executes even if Exceptions are thrown.

13.8 Exception Hierarchy

- All the main classes are in the java.lang package
- A key distinction of Exceptions is if they inherit from the RuntimeException part of the hierarchy
- Anything that inherits from Error is something that is a serious problem that an application should not try to catch
  - This means Exceptions are the most important things to consider



- The diagram above only shows the top of the hierarchy and there are many other classes that branch from Error, Exception, and RuntimeException

Exception Hierarchy

An Error usually indicates a failure of hardware or the software environment. These are things that can't be predicted and we could not even do anything about from within Java code.

13.9 Checked Exceptions

- Methods declare that an exception is possible using the ‘throws’ clause of the method declaration, e.g.

```
public void writeToStream(OutputStream s) throws
IOException {..}
```

- Instances of Exception or descendants (excluding classes derived from java.lang.RuntimeException)
- Compiler enforced
  - Code that calls a method that declares a thrown exception must either catch the exception or declare it in the method's 'throws' clause
- Both the Java compiler and the JVM check to make sure that this is obeyed

13.10 Unchecked Exceptions

- Some exception types are possible in almost any method call
- Declaring them would add too much verbosity to code; they would have to be declared as thrown in almost every method declaration
  - e.g. NullPointerException
- Every method implicitly throws RuntimeException (and descendants of RuntimeException)
- You can still catch them in a catch block if you want to

Unchecked Exceptions

Example of abnormal execution:

Integer division by zero.

Accessing an element of an array using an index larger than the size of the array.

13.11 Coding Tips - Exception Types

- The decision to use a checked or unchecked exception depends on if the client can be expected to recover
  - If the client is expected to be able to recover or handle a problem throw a checked exception

- ◇ If the client will likely not be able to recover throw an unchecked exception
- Declaring that a method throws a checked exception indicates what may go wrong and forces the client to take some action if this occurs

13.12 Catching Subclass Exceptions

- Java selects the first catch block whose caught variable is compatible with the thrown exception
- Subclass catch clauses must precede superclass catch clauses
  - ◇ Otherwise, a compiler error occurs
- If ExceptionTwo derives from ExceptionOne, the correct order of the catch clause is:

```
try {
        //…
} catch(ExceptionTwo e) {
} catch(ExceptionOne e){
} catch(Exception e) {
}
```

Catching Subclass Exceptions

The rule is that the runtime selects the first catch block whose caught variable is compatible with the thrown exception.

So a more general catch block placed in front of a specific catch block results in unreachable code.

The compiler flags unreachable code as a likely error. Remember that one of Java's goals is to catch as many errors as possible at compile time!

13.13 Java 7 – Catching Multiple Exceptions

- Prior to Java 7, every 'catch' block could only handle one type of Exception
  - ◇ This can cause some code duplication if the actions taken when different Exceptions are caught is similar

```
catch (IOException ioe) {
       logger.log(ioe);
       throw new CustomerQueryException(ioe);
} catch (SQLException sqe) {
       logger.log(sqe);
       throw new CustomerQueryException(sqe);
}
```

- In Java 7 a 'catch' block can handle more than one type of Exception
  - The types of Exceptions are separated by a '|' character

```
catch (IOException | SQLException ex) {
       logger.log(ex);
       throw new CustomerQueryException(ex);
}
```

Java 7 – Catching Multiple Exceptions

Catching multiple Exceptions can be useful when perhaps there is another catch block that does not take the same action. This makes it easy to see that the reason this catch block is still independent is because of this difference in how the Exception is handled and not just because of the limitation of having only one Exception type per catch block.

```
catch (IOException | SQLException ex) {
         logger.log(ex);
         throw new CustomerQueryException(ex);
} catch (Exception ex) {
         logger.log(ex);
         throw ex;  // This is different from the way other Exceptions are handled
}
```

13.14 Specifying Thrown Exceptions

- To specify that a method throws one or more exceptions, add a throws clause to the method signature
- The throws clause is composed of:
  - **throws** keyword
  - Followed by a comma separated list of all the exceptions thrown by the method
  - Found after the method name and argument list and before the curly braces that define the scope of the method

```
public void myMethod(int a, int b)
throws ExceptionOne, ExceptionTwo {..}
```

13.15 Rethrowing Exceptions

- If you need to pass an exception to a calling program, you can rethrow it from within the catch block using the throw statement
  - You usually do not rethrow the same exception type though

```
try {
       // code that originates an ExceptionOne
} catch(ExceptionOne e) {
       logger.error(e.getMessage());
       /* rethrow the exception
            to the calling program */
       throw e;
}
```

13.16 Java 7 – Rethrowing Exceptions

- Sometimes a catch block uses a more general Exception type to simplify the code even though the types of Exceptions caught are more specific
  - Prior to Java 7 though the method declaration would need to throw the more general type

```
public void rethrowException() throws Exception {
       try {
              // can throw FirstException or
SecondException
       } catch (Exception ex) {
              logger.log(ex); throw ex;
       }      }
```

- With Java 7 the method declaration can be changed to throw the more specific Exception types as long as the catch block uses a type that is a subtype or supertype of the catch clause exception parameter

```
public void rethrowException() throws FirstException,
SecondException {
```

```
        try {
                // can throw FirstException or
SecondException
        } catch (Exception ex) {
                logger.log(ex); throw ex;
        }       }
```

Java 7 – Rethrowing Exceptions

The importance of rethrowing the specific Exception types is now that information can be communicated to the code which calls this method instead of only getting a generalized superclass Exception object.

There are obviously a few other ways this behavior could be implemented but this allows choice to the developer in how they would want to structure Exception handling code.

With multi-catch:

```
public void rethrowException() throws FirstException, SecondException {
          try {
                    // can throw FirstException
                    // can throw SecondException
          } catch (FirstException | Second Exception ex) {
                    logger.log(ex);
                    throw ex;
          }         }
```

With multiple catch blocks (the only way prior to Java 7):

```
public void rethrowException() throws FirstException, SecondException {
          try {
                    // can throw FirstException
                    // can throw SecondException
          } catch (FirstException ex) {
                    logger.log(ex);
                    throw ex;
          } catch (Second Exception ex) {
                    logger.log(ex);
                    throw ex;
          }         }
```

13.17 Chaining Exceptions

- Often code will catch one type of exception only to throw another type of exception
  - This is known as "chained" exceptions
  - This is preferred over simply rethrowing the same exception type

- Various constructors and methods support linking one exception to another
- This allows calling code to only have to catch the outermost exception but still access additional details

```
} catch(ExceptionOne e) {
        throw new ExceptionTwo("Another message", e);
}
```

Chaining Exceptions

Note in the above example that the exception that was caught is a parameter when creating the other exception that is thrown.

This mechanism is often used to add additional details to a problem that progresses up a chain of calling methods. If the exception can't be recovered from it would be possible to print all of the details of all of the problems encountered to a log file. This can help be sure details about where the problem originated are not lost.

13.18 Creating your Own Exception

- Define in a class that extends Exception

```
public class OverdraftException extends Exception
```

- Provide whatever constructors make sense in your application

```
public OverdraftException(double currentBalance) {…}
public OverdraftException(double currentBalance,
    String message) {…}
```

- Remember that throwing an Exception is a way of communicating information
  - Have your exception object store data that the caller will find useful in dealing with the exception
    - Error message, error code, or root cause exception
- Name the exception class something that can indicate the type of problem
  - DuplicateCustomerException, DatabaseDownException, etc

Creating your Own Exception

The definition of the OverdraftException class:

```
public class OverdraftException extends Exception {
          private double currentBalance;
          public OverdraftException(double currentBalance, String message) {
                    super(message);
                    this.currentBalance = currentBalance;
          }
          public OverdraftException(double currentBalance) {
                    super();
                    this.currentBalance = currentBalance;
          }
          public double getCurrentBalance() {
                    return currentBalance;
          }
}
```

13.19 Creating your Own Exception

- Declare that your method can throw the exception
  - This is in the class that might throw the exception

```
public void withdraw(double amt) throws OverDraftException
```

- Use the method that might throw an exception inside the try block
  - This is in the class that calls code that can throw exceptions

```
try {
        a.withdraw(300);
} catch (OverDraftException e) {
        System.out.println(e.getMessage());
}
```

Creating your Own Exception

The method withdraw() of the Account class will potentially throw an OverDraftException.

```
public void withdraw(double amt) throws OverDraftException{
          if(balance < amt) {
                    throw new OverDraftException(currentBalance,
                              "Transaction Denied");
          } else {
                    balance -= amt;
          }
}
```

In the main() method of the class Driver:

```
public static void main (String args[]) {
          Account a = ("Bill", "Joy", 200);
          try {
                    a.withdraw(300);
          } catch (OverDraftException e) {
                    System.out.println(e.getMessage());
          }
}
```

13.20 Java 7 – try-with-resources Statement

- It is common to have code that opens some resources for the execution of the code and then closes these resources in a 'finally' block
  - Although this is fairly boilerplate code, implementing this pattern can be tricky and omitting a step can cause issues

```
BufferedReader br = new BufferedReader(new
FileReader(path));
try {
  return br.readLine();
} finally {
  if (br != null) br.close();
}
```

- Java 7 adds the 'try-with-resources' statement where resources can be opened within a set of parenthesis on the 'try' block and these will automatically be closed at the conclusion of the 'try' block, making the code easier to write and less error-prone

```
try (BufferedReader br = new BufferedReader(new
FileReader(path))) {
  return br.readLine();
}
```

- Multiple statements can initialize resources if separated by a ';' within the parenthesis

```
try ( ZipFile zf = new ZipFile(zipFileName);
      FileWriter writer =  new FileWriter("foo.out") ) {
```

Java 7 – try-with-resources Statement

Any resource that implements the new 'java.lang.AutoCloseable' interface. This includes all objects that also implement the 'java.io.Closeable' interface.

Note that even though this statement is called 'try-with-resources' the only difference is the set of parenthesis after the 'try' keyword and before the curly brackets for the block of code.

Of all the language changes introduced in Java 7, the 'try-with-resources' statement has perhaps the greatest potential to improve the overall quality of code as developers will no longer need to add the complex code required to make sure the resources are closed properly.

13.21 Java 7 – Suppressed Exceptions in try-with-resources

- It is possible that code that uses resources can throw an Exception during the code in the 'try' block and ALSO when the resources are closed
  - When using a 'finally' block to close the resources, it will be the Exception thrown from this block that will be returned to the code calling the method
  - This meant an extra try/catch statement was often added to make sure this Exception was ignored so the Exception from the original 'try' block was returned

```
try {
        // throws more important Exception
} finally {
  try { if (br != null) br.close();
       } catch (Exception e) { } // ignore this Exception
}
```

- When using a Java 7 'try-with-resources' statement, the Exception(s) thrown from closing the resources are "suppressed" and the Exception that would be seen by calling code is the one from the 'try' block
  - This makes the default behavior what is most often desired
  - You can obtain the suppressed Exceptions by calling the 'Throwable.getSuppressed()' method

Java 7 – Suppressed Exceptions in try-with-resources

In the code example from the slide, without the try/catch block within the finally block, the Exception that would have been returned is the one from attempting to close the resource. This is often not an important Exception because the purpose of the finally block was simply "make sure the resource is closed". The Exception that might have come from the original 'try' block is probably more important from a business logic perspective. The 'try-with-resources' statement approach makes sure that the more important Exception is what the calling code will see and not the Exception that may have

occurred from closing the resource.

Even with Java 7, if you write code using an explicit 'finally' block, the behavior will still be the same as before. You must use the 'try-with-resources' which initializes resources in parenthesis for the try block and does not have an explicit finally block to get the behavior where the Exception from closing the resources is suppressed.

13.22 Summary

- Exceptions are the mechanism used by Java to report problems
- Using exceptions can help make code more robust as it is capable of handling different problems
- A "catch" block only executes if a particular exception occurs while a "finally" block always executes
- Exceptions can be "checked" or "unchecked" depending on if they inherit from RuntimeException
- You can use the various exception classes that are part of Java or create your own that are more application-specific

Chapter 14 - Interfaces and Polymorphism

Objectives

After completing this unit, you will be able to:

- Define abstract class and interfaces
- Write classes that implement interfaces
- Understand polymorphism and how to apply the principle to write better code

14.1 Casting Objects

- A variable of an **Object** type may refer to an instance of any type
- The **Object** variable may only use the methods defined in the **Object** class
- The variable must be cast to the type of the instance, similar to converting primitive types, to use the methods of the specific class
- This is done by placing the name of the class within parenthesis in front of the variable name
  - Sometimes extra parentheses are needed to ensure the cast happens before other operations or method calls

```
Object obj = new Video("Tron");
((Video)obj).rent();     // illegal without casting
```

14.2 Casting Objects

- Explicit conversion from one data type to another
- Going up to the reference tree may be accomplished implicitly through conversion
- Going down to the reference tree requires explicit casting

Casting Objects

Object reference type changes are caused by:

Assignment conversion: assigning a variable to an object of a subclass type.

Method-call conversion: Pass in an object of a subclass type to a method parameter declared as the superclass type.

Explicit casting: Prefix the variable with a cast to a subclass type. This is the only way to go "deeper" down the type hierarchy.

14.3 The instanceof Operator

- The 'instanceof' operator is used to test the type of an object

```
public class Account extends Object {..}
public class OverDraftAccount extends Account {..}

public void testLimit(Account a) {
       if (a instanceof OverDraftAccount) {
              OverDratfAccount acct = (OverDraftAccount)a;
              double limit = acct.getOverdraftLimit();
              System.out.println("The limit is: " + limit);
       }
}
```

14.4 Abstract Classes

- A class which has its form and structure declared, without a complete implementation (e.g., missing the body of a method)
- The abstract class you define will be the superclass to a class which completes the definition
- An abstract class is not fully defined, so you cannot create an instance of it
- Abstract classes are then extended by "concrete" classes which must implement the abstract methods from the abstract class
- An abstract class is declared with the 'abstract' keyword in the class declaration
- An abstract class can have:
  - Fields

- Constructors (to initialize fields in the abstract class)
- Complete methods with an implementation
- Method declarations without an implementation and marked with the 'abstract' keyword

Abstract Classes

An abstract class does not need to contain any abstract methods. It is possible to mark the class as abstract without any abstract methods just to indicate that instances of the class can't be created.

Abstract classes are used to form inheritance hierarchies.

To create an instance of a class, all declared methods must be implemented. Using an abstract class, you can extend it in a subclass, and implement the missing parts. Once the derived class is complete, with all requirements implemented, you can use the derived class, creating instances of it.

14.5 Abstract Class – An Example

```
public abstract class Account {
	private int accountID;
	public Account(int accountID) {
		this.accountID = accountID;
	}
	public int getAccountID() {
		return accountID;
	}
	abstract public void deposit(double amount);
}

class SavingsAccount extends Account {
	public SavingsAccount(int accountID) {
		super(accountID);
	}
	public void deposit(double amount) {
		// deposit into a SavingsAccount
	}
}
```

Abstract Class – An Example

Notice that the abstract Account class does have a concrete implementation of the field and 'get' method for the accountID property. It does not have a concrete implementation for the 'deposit' method.

Note that the abstract Account class has a constructor that is called by the constructor of the extending SavingsAccount class. Constructors of abstract classes will always be called from the constructors of extending classes since you can never create an instance of an abstract class directly.

The concrete class SavingsAccount extends Account so it inherits the 'accountID' property already defined in Account. It must also implement the 'deposit' method.

14.6 Interface

- Interfaces focus on defining methods that provide the operational contract of the interface type
- Can contain fields, but these are implicitly public, static and final (constants)
- To create an interface, use the **interface** keyword
- To make a class conform to a particular interface, use the **implements** keyword
- One interface can also extend another interface

Interface

Allows for having multiple inheritance functionality.

Since these classes are abstract, they cannot be instantiated.

When a class **implements** an interface, it must complete each method definition and implementation for all methods declared in the interface.

14.7 Interface – An Example

```
public interface Animal {
       public void eat(Food toEat);
       public void move();
}
```

```
public class Dog implements Animal {
	public void eat(Food toEat) {
		// Eat like a dog    }
	public void move() {
		// Move like a dog   }
}

public class Fish implements Animal {
	public void eat(Food toEat) {
		// Eat like a fish   }
	public void move() {
		// Move like a fish  }
}
```

Interface – An Example

To create an interface, the keyword interface is used instead of class.

An interface can have:

- Fields, but they are public, static and final by default.
- Methods

Methods defined by an interface must be implemented by classes that implement the interface; otherwise, the class must be declared as an abstract class.

A class can implement multiple interfaces.

Thus,

interface A {..}

interface B {..}

then:

class newClass implements A, B {..}

14.8 Comparable Interface

- One commonly implemented interface is the java.lang.Comparable interface
  - This interface defines a 'compareTo' method often used to sort

elements in relation to each other

```
public int compareTo(Object o) {..}
```

- The following relations are required

```
x.compareTo(y)>0 && y.compareTo(z)>0 implies
x.compareTo(z)>0

x.compareTo(y)==0 implies x.compareTo(z)==y.compareTo(z)
```

- The following is suggested but not required

```
(x.compareTo(y)==0) == (x.equals(y))
```

Comparable Interface

The **compareTo** method returns the following values:

- Negative if the object the method is called on is "less than" the object passed as a parameter.
- Positive if the object the method is called on is "greater than" the object passed as a parameter.

The last relationship is often called "consistent with equals". The idea is that the equals method indicates two objects are "equal" if and only if the **compareTo** method returns a zero. Although this is not strictly required, any class that violates this with implementations of an equals and **compareTo** method should clearly document this. This type of class might behave strangely with some of the sorting algorithms provided by Java.

14.9 Comparable Example

```
public class Account implements Comparable {
	// Other definitions omitted
	public int compareTo(Object other) {
		if(!(other instanceof Account)) {
			throw new ClassCastException();
		}
		int otherID = ((Account)other).getID();
		return getID() - otherID;
}
```

Comparable Example

When providing a custom **compareTo** implementation be careful of the value returned. Returning a negative or positive value has very specific meaning for the "natural order" of the objects being compared. It is very easy to reverse the way the value is returned and accidentally reverse the sorting order from what you want it to be.

In the above example, it is implied that accounts that have a lower id would come "before" accounts with a higher id in the "natural ordering".

14.10 Java 8 – Default Methods

- One problem with modifying interfaces is that if you add a new method to an interface all existing code that had previously implemented that interface will now break because they do not implement the new method
- Java 8 added "**Default methods**" to Java interfaces
  - These allow an implementation to be provided for the method within the interface declaration itself
  - These default method implementations would be used if an implementing class did not override the method with a different implementation
- Prior to Java 8, the only way to have a Java definition where some methods were abstract and others were implemented was to have an abstract class
  - Interface methods couldn't have any method body
- Interfaces still can't have constructors or fields (state) which abstract classes can have
  - So the code in a default interface method will generally only be able to work with the parameters of the method and any other methods defined by the interface

Java 8 – Default Methods

Java 8 introduced a LOT of API changes to existing interfaces and classes. Default methods were a way to increase the backwards compatibility of code written prior to Java 8.

14.11 Java 8 – Default Method Example

- Default methods are defined with the 'default' keyword and a method body

```
public interface Vehicle {
	public void drive(int distance);
	default public void reverse(int distance) {
		drive(-distance);
	}
}

public class Car implements Vehicle {
	public void drive(int distance) {
		// drive like a car
	}
	// Don't NEED to implement reverse although we could
}
```

Java 8 – Default Method Example

In this example let's say that we want to add a new 'reverse' method to the 'Vehicle' interface. We could define the default implementation to simply call the existing 'drive' method with the negative distance. If a class implements the Vehicle interface it only HAS to implement the 'drive' method because that is abstract in the interface. Since the 'reverse' method in the interface has a default implementation, it is optional for any class that implements Vehicle to implement the 'reverse' method.

If a class implements two different interfaces and those two interfaces both provide a default method with the same signature, the implementing class MUST override that method. If it did not there would be ambiguity on which default method implementation would be called when that method is invoked on the object.

14.12 Java 8 – Static Methods

- Java 8 also added "**Static methods**"
  - These are similar to "default methods" in that they have an implementation in the interface
  - They use the 'static' keyword and are associated with the entire class and not individual instances
    - This means they could only call other static methods

```
public interface Vehicle {
	static public String[] requiredDirections() {
		return {"Forward", "Left", "Right"};
	}
	// …
}
```

14.13 Coding Tips - Superclass or Abstract Class/Interface?

- The question is when to define a superclass with various subclasses and when to define an interface or abstract class and various implementations?
- Think if it makes sense for an instance to exist of only the type in question
  - It makes no sense to say 'new Animal()' so Animal may be an interface or abstract class with various concrete implementations like Dog, Cat, etc
  - It makes sense to say 'new Account' so Account may be a class that is extended by subclasses like CheckingAccount, SavingsAccount, etc
- Also remember that a class can only extend one other class but may implement many interfaces

14.14 Coding Tips – Abstract Class or Interface

- Both abstract classes and interfaces can define methods that must be implemented by subclasses or implementations
- With default methods of Java 8, both interfaces and abstract classes can provide implementations of methods
- The main difference between the two is that abstract classes can also include fields for storing state that could be accessed by method implementation code
- Think of an 'Account' definition
  - Both an interface and an abstract class could have a method, even with an implementation, for a 'deposit(double amount)' method
  - Only an abstract class could have a field for an accountID though and

store that state at the abstract class level

- ◇ If an interface had a 'getAccountID' method without a field to store that state, implementing class would still likely have to supply code to support storing the state of the accountID

14.15 Polymorphism

- Ability to have many different forms
- An instance has only one form
- Made possible by dynamic or late binding
- If a method is called, the behavior is determined by the type of the instance, not the type of the variable
  - ◇ This means that a variable declared as an interface or superclass type may behave differently depending on the actual instance the variable refers to

```
Dog aDog = shelter.rescueDog();
aDog.bark();  // Different depending on the type of dog
```

- New objects can be added to the class hierarchy in the future

Polymorphism

Java allows you to refer to an object with a variable that is one of the parent class types; thus, you can say:

```
Account a = new OverDraftAccount();
```

Using the variable a, you can access only parts of the object that are part of the Account; the OverDraftAccount part is hidden.

```
a.setOverDraftLimit(1000.00);  // error
```

14.16 Conditions for Polymorphism

- Works with derived class objects
- The method called must be a member of the base class

- The following must be the same in the base and derived class:
  - Method signature
  - Return type
- The method access specifier must be no more restrictive in the derived class than the base

14.17 Coding Tips - Leveraging Polymorphism

- Use polymorphism wherever possible
  - This makes code more generic and adaptable to change
- Method parameters and return types should be as generic as makes sense
  - Using a class or interface further up the inheritance structure can avoid duplicating definitions

```
public class Garage {
	void park(Vehicle toPark) {..}
	/* Makes more sense than having both
		void park(Car carToPark) {..}
		void park(Truck truckToPark) {..}
	*/
}
```

- This allows for more flexible code but to also get the specific behavior defined by the actual instance

Coding Tips - Leveraging Polymorphism

The above example assumes there would be no difference as far as the Garage is concerned between parking a Car and parking a Truck. If the only interaction needed to perform the steps of the 'park' method can be written in a generic Vehicle interface it is best to use this type as the method parameter. Even though the Car and Truck classes above could have different implementations of common methods, if there is not a different sequence of steps needed to park a Car compared to parking a Truck it is better to have one method that can take either as a parameter.

14.18 Covariant Return Types

- Allows a subclass to override a superclass method and return a more specific return type

```
class A {
       public X getValue() { ... }
}

class B extends A {
       public Y getValue() { ... }
}

class Y extends X { ... }
```

- Prior to Java 5 the above wouldn't compile as you can't define two methods that differ only in the return type and prior to Java 5 the code above would not be overriding the method so the compiler would assume you were declaring a new method

Covariant Return Types

Covariant return types allows you to place restrictions on the type of object that is returned from an overridden method. The return type of the superclass can be generic but the overridden method of the subclass can be written to return a specific subtype.

14.19 Covariant Return Types – An Example

- Let's assume the following class

```
public class Garage {
       public Vehicle getVehicle() { ... }
}
```

- Prior to Java 5 you could not change the return type when overriding a method

```
public class MotorcycleGarage extends Garage {
       public Vehicle getVehicle() {
              return new Motorcycle(); }
}
```

```
MotorcycleGarage garage = new MotorcycleGarage();
Motorcycle vehicle = (Motorcycle) garage.getVehicle();
```

- Covariant return types in Java 5

```
public class MotorcycleGarage extends Garage {
       public Motorcycle getVehicle() { ... }
}

MotorcycleGarage garage = new MotorcycleGarage();
Motorcycle vehicle = garage.getVehicle();
```

Covariant Return Types – An Example

Prior to Java 5 even though the MotorcycleGarage implementation of getVehicle() can return a Motorcycle object the method declaration can only declare that it returns a Vehicle. Even though we know we might be working with a MotorcycleGarage the code that calls the method has to cast the object returned.

With Java 5 covariant return types the declaration of the getVehicle() method in the MotorcycleGarage subclass can declare it will only return Motorcycle objects and is still overriding the superclass implementation of the method. This allows the code that calls the method to use a more specific type without needing to cast.

14.20 Summary

- Both abstract classes and interfaces define methods
- Some methods can have an implementation and other methods marked as abstract without implementation
- Java 8 added default and static methods allowing interface method declarations to contain a method implementation
- The primary difference now between abstract classes and interfaces is that abstract classes can contain fields to store state
- Polymorphism can help make adaptable code as the behavior of specific instances can be decided by the object type and not the type of the variable

Chapter 15 - Useful Java Classes

Objectives

After completing this unit, you will be able to:

- Understand the logging capability of the Java Logging API
- Use various classes when working with Strings, Dates, and formatting

15.1 Java Logging API

- Contained in java.util.logging package
- Generate log messages for users performing different roles, such as administrators and developers
- Designed to have minimal performance impact
- Facility to generate different levels of logging

Java Logging API

Logging API: The Java Logging APIs facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams. The Logging APIs capture information such as security failures, configuration errors, performance bottlenecks, and/or bugs in the application or platform.

The logging API is designed to let java programs generate messages of interest to end users, system administrators, developers and trouble shooters.

The logging API is designed to have a minimal performance impact on production systems. The API provides various levels of logging to provide greater control.

15.2 Control Flow of Logging

- Application code generally only works with Logger objects to send log messages

Control Flow of Logging

Applications make logging calls on **Logger** objects. Loggers are organized in a hierarchical namespace and child Loggers may inherit some logging properties from their parents in the namespace.

Applications make logging calls on **Logger** objects. These Logger objects allocate **LogRecord** objects which are passed to Handler objects for publication. Both Loggers and Handlers may use logging **Levels** and (optionally) **Filters** to decide if they are interested in a particular **LogRecord**. When it is necessary to publish a **LogRecord** externally, a Handler can (optionally) use a Formatter to localize and format the message before publishing it to an I/O stream.

The recommended approach as per the Logging API is for the logger to have the same name as the java class or the package.

15.3 Logging Levels

- It is desirable to be able to control the level of logging without changing the code directly
  - This allows staging and production systems to log only high priority messages like errors or warnings but development systems to log more detail
- To help achieve this, the logging system defines a class called Level to specify the importance of a logging record
- The logging library has a set of predefined logging levels:
  - SEVERE - The highest value; intended for extremely important messages (e.g. fatal program errors)
  - WARNING - Intended for warning messages
  - INFO - Informational runtime messages

- CONFIG - Informational messages about configuration settings/setup
- FINE - Used for greater detail, when debugging/diagnosing problems
- FINER - Even greater detail
- FINEST - The lowest value; greatest detail

Logging Levels

Logging level can be set using either:

- Logging configuration file
- Calling Logger.setLevel method

If the request level is less than log level (of Logger), then logging call returns immediately (i.e., logging is not performed).

15.4 Loggers

- Logger objects receive log requests from client code
- Each logger keeps track of the log level specified. Log requests below that level are discarded
- Also possible to create anonymous loggers that don't appear in shared namespace
- Loggers are managed by the Log Manager

Loggers

Logger inherits its parents':

- Level – If child's logging level is not set or if it is set to null
- Handlers – Child publishes log records to the parent's handlers as well
- Resource Bundles – If child doesn't have its own (used for localizing messages)

15.5 Logging Example

- Import the logging package

```
import java.util.logging.Logger;
```

- Obtain a reference to the logger for the class

```
private Logger logger =
Logger.getLogger(this.getClass().getName());
```

- Within code make calls to the various methods of the logger

```
logger.severe("Customer was not registered");
logger.info("Opening file");
logger.finer("DEBUG: Customer details: " + cust);
```

Logging Example

The first two steps could be the same in every class that uses logging.

It is up to the code you write to decide what types of messages should be output and what level they should be set at. The power of the logging API is the ability to "tune" the level of messages to what is appropriate for a given environment. If you only use 1-2 of the levels available you will have less choice to change what levels of detail you see.

15.6 Logging Handlers

- A handler stores log messages in a specific device:
  - StreamHandler: A simple handler for writing formatted records to an OutputStream.
  - ConsoleHandler: A simple handler for writing formatted records to System.err.
  - FileHandler: A handler that writes formatted log records either to a single file, or to a set of rotating log files.
  - SocketHandler: A handler that writes formatted log records to remote TCP ports.
  - MemoryHandler: A handler that buffers log records in memory.

15.7 Logging Formatters & Log Manager

- Formatters

- SimpleFormatter: Writes brief "human-readable" summaries of log records
- XMLFormatter: Writes detailed XML-structured information

- Log Manager contains
  - A hierarchical namespace of named Loggers
    - Usually every Java class gets a separate logger.
    - Logging configuration can be set at the package level or specific class level.
- A set of logging control properties read from the configuration file

Logging Formatters & Log Manager

Developers can provide added functionality by subclassing the Handler or Formatter classes.

The LogManager keeps track of the global logging information. Configuration parameters can be specified as:

- Command line arguments
- Parameters compiled into program
- Runtime configuration file

15.8 Logging Configuration File

- The logging configuration can be initialized using a logging configuration file that will be read at startup
  - This logging configuration file is in standard **java.util.Properties** format
  - The initial configuration may specify levels for particular loggers. These levels are applied to the named logger and any loggers below it in the naming hierarchy
- By default, the LogManager reads the configuration from the 'lib/logging.properties' file in the Java runtime that is running the Java program
  - You can also set the '**java.util.logging.config.file**' system property to specify an alternate file for configuration

- Alternatively, you can load a configuration file from any location using the **LogManager.readConfiguration(InputStream ins)** method

15.9 Example Logging Configuration File

- Store log messages in a file

```
handlers= java.util.logging.FileHandler
```

- Specify file logging properties

```
java.util.logging.FileHandler.pattern =
logs/myapp.log%g.log
```

- Create a new log file every 5MB

```
java.util.logging.FileHandler.limit = 5242880
```

- Keep only three recent log files. Delete older log files.

```
java.util.logging.FileHandler.count = 3
java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter
```

- Set the default log level to INFO

```
.level= INFO
```

- Override the log level for a specific package – com.webage.utils

```
com.webage.utils.level = SEVERE
```

Example Logging Configuration File

By using the pattern logs/myapp.log%g.log, three log files will be generated with names such as myapp0.log, myapp1.log and myapp2.log, each of which will contain up to 5MB of logs. If the pattern was logs/myapp.log, then the same log file would be over written upon reaching a file size of 5 MB.

15.10 Logging Filters

- Sometimes the level of a log record may not give enough control about whether or not to log the record

- The logging API defines a Filter interface that can be implemented and associated with a Logger or Handler
  - This interface includes only one method

```
public boolean isLoggable(LogRecord record)
```

- If you implement this interface you can use the setFilter method on a Logger or Handler

Logging Filters

The Java logging API does not provide any implementations of the Filter interface. It is present to allow for extension of the logging mechanism.

15.11 java.lang.StringBuilder

- Can be modified directly, unlike a String
- Contains a "buffer" of allocated space. This is the capacity
- Increases automatically
- The length is the number of characters contained
- Better performance when modifying StringBuilder objects rather than Strings

- Common Methods:

```
capacity()
length()
ensureCapacity(int)
setLength(int)
substring(…)
reverse()
append(…)
delete(int, int)
deleteCharAt(int)
replace(int, int, String)
setCharAt(int, char)
insert(…)
```

java.lang.StringBuilder

The setLength(int) method can alter the contents of the StringBuilder. If the argument passed to the method is less than the length, any characters which do not fit within the new length are truncated and lost. If the argument is greater than the current length, null characters ('\u0000') are appended to the StringBuilder until the length equals the value of the argument.

Java can run code concurrently in multiple "threads" of execution. If multiple "threads" need to alter the same sequence of characters it is more appropriate to use the StringBuffer class which is "thread

safe". Full discussion of multi-threaded programming is outside the scope of this lecture but being "thread safe" is something to consider when using various Java classes using this multi-threaded model.

Internally Java uses StringBuffer objects to perform the concatenation of String objects. It is this overhead that makes String objects inefficient if they are modified or combined fairly frequently.

15.12 java.util.StringTokenizer

- Used to break Strings into "tokens"
  - A "token" is the set of characters between delimiting characters
  - For example, breaking a sentence into words based on spaces
- Delimiting characters can be specified
- Methods to extract and detect tokens
  - Common Methods:

```
countTokens()
hasMoreTokens()
nextToken()
nextToken(String)
```

  - Useful Constructors:

```
StringTokenizer(String)
StringTokenizer(String,
String)
```

java.util.StringTokenizer

All constructors of StringTokenizer objects require that you pass in the String to be tokenized. The constructor above which takes only the String to be tokenized uses default characters for delimiting tokens. These default characters are: space, tab, newline, carriage-return, and form-feed characters. The constructor above which takes two String arguments can be used to indicate characters used to delimit tokens. The characters which make up the second String argument are used to delimit tokens. A String of " ," would delimit a String based on spaces or commas.

By default, delimiting characters are not returned as part of tokens or as individual tokens from a StringTokenizer. The following code would return a comma on the end of the second token:

```
String withCommas = "For example, this line has commas";
StringTokenizer st = new StringTokenizer(withCommas);
while (st.hasMoreTokens()) {
          System.out.println(st.nextToken());
}
```

This code would remove the comma from the end of the second token:

```
String withCommas = "For example, this line has commas";
```

```
StringTokenizer st = new StringTokenizer(withCommas, " ,");
while (st.hasMoreTokens()) {
          System.out.println(st.nextToken());
}
```

15.13 java.util.Arrays & java.util.Collections

- These classes contain a number of static utility methods that can operate on arrays or collections
- Generally this includes functions to sort, search, and "fill" an array or collection
  - Shared common methods:

```
binarySearch(..)
fill(..)
sort(..)
```

  - Collections methods:

```
copy(..), reverse(..)
max(..), min(..)
replaceAll(..)
swap(..)
```

java.util.Arrays & java.util.Collections

The binary search mentioned above generally assumes the array or collection is sorted before performing the search.

15.14 java.util.Random

- Provides a way to generate "pseudorandom" numbers based on a "seed"
  - ◇ You can set the seed in one of the constructors or a 'setSeed' method
- There are various methods to return the "next" random number for all primitive numeric types
  - ◇ This can be more useful than **Math.random()** which only returns a double
  - ◇ You can also obtain a "Gaussian" distribution

```
Random generator = new Random();
int[] randomInts = new int[100];
for (int i = 0; i < randomInts.length; i++) {
```

```
            // random ints between zero and 100
            randomInts[i] = generator.nextInt(100);
      }
      generator = new Random();
      double[] gaussian = new double[100];
      for (int i = 0; i < gaussian.length; i++) {
            gaussian[i] = generator.nextGaussian();
      }
```

java.util.Random

The reason that this is called "pseudorandom" numbers is because it is true that two Random objects constructed with the same "seed" will return the same sequence of numbers. If you construct a Random object without supplying the seed, the seed is set to something very likely distinct from every other call to the constructor giving you a "random" number generator.

The implementation of the Math.random method uses the Random class to generate the numbers returned. The first call to the Math.random method creates the Random object which is used for all subsequent calls to the Math.random method while the program is executing.

15.15 Java Date and Time

- Over the history of Java there have been a few ways to work with dates and times
  - JDK 1.0 – **java.util.Date** class used for pretty much everything
  - JDK 1.1 – **java.util.Calendar** to provide a more generic date system replacing some things from the Date class
  - JDK 1.8 – Completely new and more robust **java.time** package for more advanced usage
- Although Java has always had ways of working with date and time, there were issues with the previous approach
  - No fluent API approach
  - Instances are mutable
  - Not thread safe
  - Weakly typed calendars
  - One size fits all (java.util.Date is for date AND time)

15.16 Local Date and Time

- The **java.time** package has 3 main classes for working with “local” dates and times (without a time zone)
  - LocalDate
    - Date but not time
    - Year-Month-Day representation
    - **toString** – ISO 8601 format (YYYY-MM-DD)
  - LocalTime
    - Time but not date
    - Hours-minutes-seconds-nanoseconds representation
    - **toString** – HH:MM:SS
  - LocalDateTime
    - Combination of the other 2
    - Useful for events
- These classes don't have constructors so you use static methods to get an instance of the desired type

15.17 java.util.Date and java.time.Instant

- **java.util.Date**
  - Represents a specific instant in time
  - The default constructor creates an object that represents the instant in time when it was created
  - Contains 'before' and 'after' methods for finding out the relative position of two Date objects
  - Most of the methods defined in this class are "deprecated" in favor of other methods defined in the Calendar and DateFormat classes
- **java.time.Instant** – Stores an instant in time on the timeline
  - Closest equivalent to java.util.Date

- Stored as seconds (long) and nanoseconds (int)

java.util.Date and java.time.Instant

The notion of "deprecation" in Java means that even though something may still work there are newer, better ways to do things. The documentation of the Date class will indicate the classes and methods that should be used for the newer functionality.

15.18 Formatting

- There are various classes in the **java.text** package that can format output
  - DateFormat & SimpleDateFormat
  - NumberFormat & DecimalFormat
  - MessageFormat
- You can also use the **java.util.Formatter** class for **printf**-style formatted output
  - Many classes provide convenience methods for this style of formatting
- This formatting can be "locale-sensitive" so you can easily write code that will behave as expected under different cultural conventions
- Most of the time you use a '**getInstance**' method to obtain a formatter for the default locale or a specified locale
- You can use special characters with **SimpleDateFormat** and **DecimalFormat** to control how the output is formatted
- There is **java.time.format.DateTimeFormatter** for use with the newer Java 8 API
- More details on how to use these classes to control formatting are available in the Java API documentation

15.19 Formatting Example

```
NumberFormat nf = NumberFormat.getCurrencyInstance();
String output;
output = nf.format(account.getBalance());
```

```
DateFormat df =
DateFormat.getDateInstance(DateFormat.MEDIUM);
String dateOut;
dateOut = df.format(new Date());

System.out.println("Balance: " + output);
System.out.println("Today's date: " + dateOut);
```

Formatting Example

The example above would provide output similar to:

Balance: $16,254.48

Today's date: Oct 9, 2016

15.20 Summary

- The java.util.logging package provides a robust logging capability
- There are various classes to work with sequences of characters and Strings
- Classes like Date and Calendar provide a way to work with dates and times
- Using the various Format classes can help write code that produces locale-sensitive output

Chapter 16 - Collections and Generics

Objectives

After completing this unit, you will be able to:

- Use collections to store and retrieve objects
- Understand the structure of the collections framework
- Describe some of the major classes and interfaces in the collections framework
- Use collections effectively with generics and autoboxing

16.1 What are Collections?

- Pre-built framework for storing objects in memory
- Largely removes the need to implement data structures like resizable arrays, linked lists, etc.
- Framework classes cover the most common storage scenarios
  - Most of the classes and interfaces are in the **java.util** package
- Extensible for other situations

What are Collections?

This chapter will not attempt to provide an exhaustive reference for the Collections APIs but simply provide an overview. Refer to the Java API reference documentation for more details.

16.2 Arrays vs. Collections

- Arrays
  - Created with a fixed size
  - Elements accessed using special syntax
  - All elements are of the declared type of the array
- Collections
  - Can increase size on demand
  - Elements accessed using methods
  - Elements may be of different types

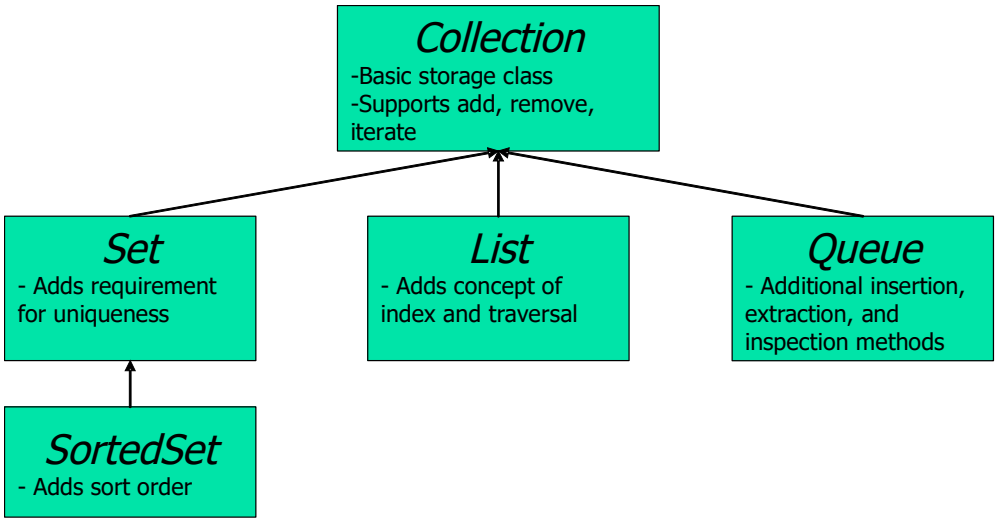
- Can store primitives or Objects
- Only stores Objects

Arrays vs. Collections

Some of the similarities between Arrays and Collections include the concept of element indices as well as using one variable to refer to the entire collection.

Although the fact that Collections can only store Objects would seem like a restriction this will be addressed later in the chapter with a feature called "autoboxing".

16.3 Main Collections Interfaces



16.4 java.util.Collection

- The framework doesn't provide any direct implementations of **Collection**
  - It just defines the basic operations
  - Classes generally implement more specific interfaces like **Set** and **List**
- Method Parameters and return values should be declared as type **Collection** or one of the other interface types for maximum generality

16.5 Main Collection Methods

```
boolean add(Object o)
void clear()
```

```
boolean contains(Object o)
boolean isEmpty()
Iterator iterator()
boolean remove(Object o)
int size()
Object[] toArray()
```

Main Collection Methods

The boolean value returned by the add and remove methods indicate if the collection changed as a result of the method call.

16.6 Sets

- **java.util.Set**
  - Doesn’t add any methods, but indicates that the concrete class’s behavior is constrained to be a set
    - Requires that there are no duplicates
    - Formally, no pair of elements e1 and e2 such that e1.equals(e2)==true, and maximum of one null element
- **java.util.SortedSet**
  - Declares that the set will be sorted, either according to the natural order of elements or a comparator
  - Allows you to retrieve the comparator that sorts the set
- **java.util.NavigableSet**
  - A SortedSet that also lets you search for elements that are "closest" to a given search target
  - You can also remove and return the first or last element with one operation

Sets

Any objects you store need to provide a sensible implementation of the equals() method.

16.7 java.util.List

- Adds the concept of position in the list
- Provides direct access to list members

```
void add(int index, Object o)
Object get(int index)
Object remove(int index)
Object set(int index, Object o)
ListIterator listIterator()
```

java.util.List

Even though the List interface provides direct access with additional methods, some implementations will perform poorly if you use these methods. For instance, a LinkedList will need to traverse the list to the required index, which will be a relatively slow operation with O(n) performance.

16.8 java.util.Queue

- Designed for holding elements prior to processing
- Provide additional operations for inserting, removing, and inspecting elements
  - Each operation type exists in two forms depending on whether the method returns a special return value or throws an Exception when the operation "fails"
- Used when it may be fairly common for these operations to "fail" and throwing an Exception is not desired
- Also used when the collection has a "capacity"

operation	throws Exception	returns special value
Insert	boolean add(Object o)	boolean offer(Object o)
Remove	Object remove()	Object poll()
Examine	Object element()	Object peek()

- Also a **java.util.Deque** interface which is a "double ended queue" that supports operations on the "head" and "tail"

java.util.Queue

You might use some of the methods on the right of the table above if you are doing operations on a queue repeatedly and don't want an exception to be returned every time the operation does not complete normally. One example of this is a loop that uses the 'poll' operation to see if anything is in the queue and then pauses for a few seconds before checking again. The 'poll' operation would simply return 'null' while the 'remove' operation would throw an exception.

Even though in English the word "queue" implies a first-in-first-out (FIFO) order, queues in Java are not required to implement that type of ordering. Other possible orderings include some type of ordering by priority or a stack (last-in-first-out).

16.9 Iteration on a Collection

- A common operation on elements of a collection is to traverse, or iterate over the collection
- This can be done in two ways
  - Using the for-each loop
  - Using an **Iterator** obtained from the collection

16.10 Iteration on a Collection

- The for-each loop is used well with a collection

```
Collection c = ...;
for (Object o : c) {
       // Do something with each Object
       System.out.println(o);
}
```

16.11 Iteration on a Collection

- **java.util.Iterator**
  - Provides functionality to iterate over the contents of a collection
  - Can also remove or replace the last element returned
- Usual code:

```
Iterator it=aCollection.iterator();
while (it.hasNext()) {
       Object var=it.next();
       /* Do something with the element. */
       System.out.println(var);
       if (var instanceof Account) {
              it.remove();
       }
}
```

16.12 Iteration on a Collection

- **java.util.ListIterator**
  - Allows you to traverse a list bidirectionally
  - Allows you to add or remove elements from the list during traversal
  - To get the performance advantage of a LinkedList (“better if you need to add or remove elements from the middle of a list”), you need to use ListIterator

16.13 Iterator vs. For-Each Loop

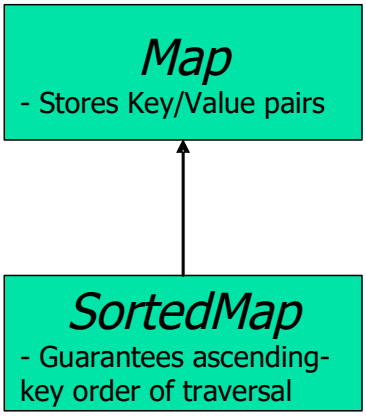
- The for-each loop is preferred when possible as it makes code simpler and less likely to contain errors
  - Nested iteration is particularly hard to handle but the for-each construct handles this well
- Using an iterator is still required if:
  - You need to remove or replace the current element while iterating through the collection
  - You need to iterate over multiple collections in parallel
    - This is different than nested iteration

Iterator vs. For-Each Loop

All iterators can remove an element during traversal. Only a ListIterator can replace elements during the traversal.

16.14 Maps

- An Object that stores key/value pairs
  - Can't contain duplicate keys
  - Each key maps to at most one value
- **Don’t use mutable objects as keys!**
  - At least don’t change objects once they’ve been used as keys
  - Behavior is undefined and probably unexpected
  - The objects stored in a map can be modified but the keys should not be



Maps

Mutable objects are those that can be changed after they are constructed. Objects like Strings are immutable.

If the key that is used for placing an object in a Map is changed after being used as a key the Map may not locate the object stored when asked if the key represents an object in the Map.

16.15 java.util.Map

- Main methods:

```
void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
Object get(Object key)
Object put(Object key, Object value)
Object remove(Object key)
```

- Can also access its contents using Collection views
  - **keySet()** returns a **Collection** containing the stored keys
  - **values**() returns a **Collection** containing the stored values
  - **entrySet**() returns a **Collection** containing **Map.Entry** objects that contain the key/value pairs

java.util.Map

The 'put' and 'remove' methods return the Object that was previously associated with the key or 'null' if there was no mapping for the key.

16.16 Other Maps

- **java.util.SortedMap**
  - Like a **Map** except that collections obtained by **keys**(), **values**() and **entrySet**() will be in ascending order of keys
  - Other methods:

```
Object firstKey()
Object lastKey()
SortedMap headMap(Object toKey)
SortedMap subMap(Object fromKey, Object toKey)
SortedMap tailMap(Object fromKey)
```

- **java.util.NavigableMap**
  - A SortedMap that also lets you search for elements that are "closest" to a given search target
  - You can also remove and return the first or last element with one operation

16.17 Collections Implementations

- **java.util.HashSet** and **java.util.HashMap**
  - Implements Set or Map using a Hashtable
  - Requires that the contained objects have reasonable implementations

of **hashCode**() and **equals**()

- Dynamically sized, but can be constructed with a given initial capacity

16.18 Collections Implementations

- **java.util.ArrayList**
  - Implements **List** using an array that will be re-created as the size of the list increases
  - Dynamically sized, but can be constructed with a given initial capacity
  - Optimal for random access of the list
  - Adding or removing elements within the list may trigger large copy operations

16.19 Collections Implementations

- **java.util.LinkedList**
  - Implements **List** and **Queue** using a doubly-linked list of entries
  - Faster than ArrayList for adding and removing entries within the list
    - Remember that traversing to the entry may take time, so for best performance use **ListIterator** approach
- **java.util.TreeMap** and **java.util.TreeSet**
  - Provides a Tree-based implementation of NavigableMap or NavigableSet
  - Guarantees log(n) cost of various operations

16.20 Abstract Implementations

- There are also some abstract classes that provide a partial implementation of some of the Collections interfaces
  - **AbstractCollection**
  - **AbstractList**

- **AbstractSequentialList**
- **AbstractMap**
- **AbstractSet**
- **AbstractQueue**

- To provide a custom implementation it may be easier to extend one of the abstract classes and override the methods that require custom behavior in addition to abstract methods not implemented in the superclass being extended

16.21 Choosing a Collection Type

- The declared type of a variable, method parameter, or return type should be of one of the interfaces and be determined by what properties the group of objects will have
  - Objects with a key would be some type of Map
  - Objects that will be unique but without keys would be some type of Set
  - If the group has some concept of "position" but no keys and not necessarily a "sorted" order you may want a List
  - For a group without any of the above properties you can just use Collection
- The implementation used will be determined by the types of objects stored and what types of operations (searching, iterating, etc) might be performed
  - If you are storing objects in "natural sorted order" you would use TreeMap or TreeSet
  - If you need to have predictable and repeatable iteration order but without using "natural sorted order" you would use some type of LinkedXXX implementation
  - If the objects you are storing have an implementation of hashCode() you may get better performance with an implementation that uses hashing

16.22 Generics

- Added in Java 5
- Allows for generic APIs to be written that can be further parameterized at compile time
  - This eliminates the need to create two APIs that only differ in the types used
- In the API you will often see variables used to indicate type parameters

```
Class ArrayList<E>
       boolean add(E element)
       E get(int index)
```

Generics

In the example above, any type substituted for the variable 'E' will be applied throughout the rest of the API for that specific usage. For example, ArrayList<String> would behave as:

```
Class ArrayList<String>
         boolean add(String element)
         String get(int index)
```

16.23 Generics and Collections

- Define collections in terms of generic types
  - Allows the compiler to enforce what types of instances can be stored in the collection
  - Eliminates the need to explicitly cast when retrieving from a collection
    - You get back the parameterized type used in the declaration
- e.g. List of Strings

```
List<String> names=new ArrayList<String>();
```

- e.g. Map of User instances indexed by String

```
Map<String, User> userMap=new HashMap<String, User>();
```

Generics and Collections

You can also define your own Generic APIs that can be parameterized but using generics with collections is by far the most used application. Defining your own generic types is beyond the scope of this lecture.

16.24 Generic Collection Example

- Generics can simplify collection code
- Old way:

```
ArrayList list = new ArrayList();
list.add("Bob");
String s = (String)list.get(0);
```

- New way:

```
ArrayList<String> list = new ArrayList<String>();
list.add("Bob");
String s = list.get(0);
```

Generic Collection Example

Even though the code above doesn't look much simpler in terms of characters or lines typed the removal of the casting every time you get an element out of the collection simplifies things when many instances are stored in the collection. The only line that has more characters typed is the first line for declaring the collection which is only done once.

16.25 Generic Collection Example

- Generics also enforce at compile time what types can be stored and retrieved
- Old way:

```
// This generates runtime exception
ArrayList list = new ArrayList();
list.add("Bob");
Customer c = (Customer)list.get(0);
```

- New way:

```
// This won't compile, even with the casting
ArrayList<String> list = new ArrayList<String>();
list.add("Bob");
Customer c = (Customer)list.get(0);
```

Generic Collection Example

Since it is always better to learn at compile time rather than runtime that code won't behave as expected generics make collection coding much more robust. It also makes it much more readable as you explicitly state what type of instance should be stored in the collection and the compiler can enforce this.

16.26 Collections and Primitive Types

- Collections only contain Objects
- Previously it was more difficult to work with collections and primitive types
  - You had to "wrap" and "unwrap" a primitive in a "wrapper" class that turned the value into an Object
- Java 5 added a feature, called "autoboxing" that make this much easier
  - The compiler can automatically convert between primitive types and the corresponding "wrapper" class
- Old way:

```
ArrayList list = new ArrayList();
list.add(new Integer(42));
int i = ((Integer)list.get(0)).intValue();
```

- New way:

```
ArrayList<Integer> list = new ArrayList<Integer>();
// Autoboxing
list.add(42);
Integer i = list.get(0);
```

Collections and Primitive Types

The use of generics above is required to get the features provided by autoboxing.

Even though with autoboxing it appears the collection is storing the primitive value directly it is still storing an object. You are simply not writing the code to convert back and forth. You could store a primitive type using autoboxing and then get the wrapper type from the collection as shown below and vice versa.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(42);
Integer i = list.get(0);
int j = list.get(0);
```

16.27 Generic Diamond Operator

- Before Java 7, to use generics in a type-safe declaration you needed to use them on both sides of the statement
  - This included the variable declaration and calling the constructor
  - This is repetitive

```
Map<Integer, List<Customer>> newCustomers =
       new TreeMap<Integer, List<Customer>> ();
```

- Java 7 adds the new "Diamond operator" or '<>' which can be used on the right-hand side and the compiler will infer what the types will be
  - The result of the code is exactly the same but easier to write

```
Map<Integer, List<Customer>> newCustomers =
       new TreeMap<> ();
```

Generic Diamond Operator

If you were to leave off generic qualification on the constructor altogether you would get compile warnings because the code would not be type-safe. The diamond operator makes sure the code is still type-safe.

16.28 Summary

- There are a number of data structure and storage mechanisms available with the collections API
- Collections are typically preferred over arrays
- Different collection implementations may perform differently depending on the type of usage
- Language enhancements like the for-each loop, generics, and autoboxing

can make it easy to write good code with collections

Chapter 17 - Introduction to Lambda Expressions

Objectives

Key objectives of this chapter

- Introduction to Lambda Expressions
- Benefits of Lambda Expressions
- Using Lambda Expressions instead of anonymous inner classes

17.1 Functional Interface

- It is very common in Java to have a “functional interface”
  - These are interfaces that have a single abstract method to implement
- Common pattern in many Java programs
  - GUI listeners
  - java.lang.Runnable
  - java.util.Comparator
- Java 8 even introduced a '@FunctionalInterface' annotation but this is mainly to make sure the compiler throws an error if the interface has more than one abstract method

```
@FunctionalInterface
public interface Comparator<T> {
       public int compare(T o1, T o2) …; }
```

- Although the declaration of these interfaces is fairly easy, the implementation of them can be tricky with some complex syntax

Functional Interface

Implementations of a functional interface can be called “SAM Classes” for single abstract method.

An interface does not need to have the '@FunctionalInterface' annotation to be considered a functional interface. This annotation just directs the compiler to thrown an error if the interface tries to declare more than one abstract method. Interfaces without the annotation are still treated as functional interfaces if they only declare a single abstract method.

Java 8 adds “default methods” which provide a method and a default implementation as part of the interface. An interface can still be a functional interface even if it has default methods as long as it has only one abstract method.

17.2 Anonymous Inner Class (AIC)

- One way to implement a functional interface is with an 'Anonymous Inner Class' (AIC)
  - This provides an implementation “on the fly” where it is needed
  - Also avoids creating lots of extra classes where reuse is not needed
- When declaring an instance of an AIC, you use the 'new' keyword but then supply the class definition inline
  - You also use the name of the interface you are implementing instead of a unique class name

```
Comparator<Person> sorter = new Comparator<Person>() {
       public int compare(Person pers1, Person pers2) {
              return pers1.getLastName().compareTo(
                     pers2.getLastName());
       }
};
```

Anonymous Inner Class (AIC)

The above example assumes a Java class 'Person' which has some basic properties like first and last name, phone, etc.

17.3 Downside of AIC

- Although the AIC does prevent the need for declaring a separate, unique class just for implementing the interface, there are a number of downsides:
  - Using standard code formatting, there end up being lots of lines of code in vertical space so it is not very compact
  - The syntax can be tricky to code or follow, especially with the addition of the curly brackets for the class and method bodies
  - Declaring an AIC as a method parameter is even trickier as you then

also have the parenthesis around the AIC for the method parameters

- ◇ You can't reuse the class (although we would have declared a separate class anyway if we needed to)
- ◇ When compiled, there are extra .class files generated with cryptic names

- Most importantly that seems like a lot of extra code just to say “compare the last names to sort”!
  - ◇ This is even worse since there is only a single method we need to implement!

Downside of AIC

The readability of an AIC can be especially tough for those that might be maintaining or debugging code they did not write. It can take a while to even figure out what is being declared with the AIC.

17.4 Lambda Expressions

- Java 8 introduced “**Lambda Expressions**” to provide much simpler syntax of functional interfaces
  - ◇ All of the extra “stuff” that was required for the syntax of an AIC is stripped away and only the implementation of the method is left
- The previous AIC

```
Comparator<Person> sorter = new Comparator<Person>() {
       public int compare(Person pers1, Person pers2) {
              return pers1.getLastName().compareTo(
                     pers2.getLastName());
       }
};
```

- is turned into this as a Lambda expression!

```
Comparator<Person> sorter = (Person pers1, Person pers2)
   -> pers1.getLastName().compareTo(pers2.getLastName());
```

Lambda Expressions

In the AIC code above, the critical parts are bold. These are the parts that are brought over into the Lambda expression. Everything else was just present for the syntax of implementing the AIC.

17.5 Lambda Expression Syntax

- There are 3 major elements of a Lambda expression

Argument list	Arrow token	Body
(int x, int y)	->	x + y

- For Lambda expressions with a single statement that return a value, the 'return' keyword is optional
- The parenthesis are also optional if there is only one parameter
- Since Lambda expressions implement a single method, the type of the parameters can be left off although having them improves readability

```
(x, y) -> x + y
```

- Lambda expressions can have multiple statements in the body in a block

```
(Person pers1, Person pers2) ->
       { String last1 = pers1.getLastName();
         String last2 = pers2.getLastName();
         return last1.compareTo(last2); }
```

Lambda Expression Syntax

Depending on the Lambda expression, several elements are optional to simplify the syntax.

17.6 Method Reference

- In some cases a Lambda expression simply calls a class method

```
pers1 -> pers1.printSummary()
```

- Instead you can use a method reference

```
Person::printSummary
```

- You can use a method reference in the following situations:
  - Reference to a static method

    ```
    ClassName::staticMethodName
    ```

  - Reference to an instance method

    ```
    objectReference::methodName
    ```

  - Reference to an instance method of an arbitrary object of a particular

type

```
Person::printSummary
```

- Reference to a constructor

```
ClassName::new
```

Method Reference

Method references are especially useful when passing a parameter to a method. Instead of having the syntax of the Lambda expression in the parameters of the method you can simply reference an existing method.

17.7 Benefits of Lambda Expressions – An Example

- To demonstrate the benefits of Lambda expressions, we will use an example of having a list of Person objects and needing to determine which instances meet certain criteria
- The big issue is where to define the code that determines the conditions that are being checked and action that is taken on the instance
  - Since there could be many different conditions that may be used, we want to avoid needing separate methods for each combination
  - It will be best to have a way to define the criteria “on demand” where the list of instances is being worked with
- We'll use three classes for the example
  - Person – Generic class for properties like first & last name, age, phone
  - PersonContact – A class for performing certain actions to contact people
  - ContactTest – A class that will run code using the Person instances

17.8 Initial Version

- In the initial version the logic for the criteria is in the PersonContact class along with the action that is taken
  - We would need every combination of criteria and action possible

```
public class ContactTest {
   public static void main(String[] args) {
      List<Person> people = // get a list of instances
      PersonContact contact = new PersonContact();
      System.out.println("Calling drivers");
      contact.callDrivers(people);
}  }

public class PersonContact {
   public void callDrivers(List<Person> people) {
      for(Person p : people) {
         if(p.getAge() >= 16) {
            callPerson(p);
}  }  }  }
```

17.9 Refactor Criteria Into Method

- Refactoring the criteria into a separate method would help with maintaining the code but would not eliminate the need to have all combinations of criteria and action in the PersonContact class

```
public class PersonContact {
   public void callDrivers(List<Person> people) {
      for(Person p : people) {
         if(isDriver(p)) {
            callPerson(p);
}  }  }  }

private boolean isDriver(Person p) {
   return p.getAge() >= 16;
}
```

Refactor Criteria Into Method

Even though the code above looks simple enough, imagine if we had more actions that could be taken for contacting people (calling, emailing, and mailing) and more criteria for the groups (drivers, retirees, and women). Even for these few combinations we would need 9 methods, not very maintainable.

17.10 Predicate Interface

- What is needed is to have an interface that defines a method to return a boolean based on some test
  - This could let us factor the criteria for the test out of the PersonContact class

```
public interface MyTest<T> {
   public boolean test(T t);
}
```

- The reality is that Java 8 added a new '**java.util.function.Predicate**' interface that is exactly this

```
@FunctionalInterface
public interface Predicate<T> {
   public boolean test(T t);
}
```

17.11 Using a Predicate

- If the methods for the actions on the PersonContact class take a Predicate as a parameter, the criteria for which Person the action should apply to can be factored out to the code calling the method

```
public class PersonContact {
   public void callPeople(List<Person> people,
      Predicate<Person> criteria) {
      for(Person p : people) {
         if(criteria.test(p)) {
            callPerson(p);
}  }  }  }
```

- So how to provide an implementation of Predicate from the calling code?

17.12 Implement as Separate Class

- The Predicate interface could be implemented as a separate concrete class

```
public class DriverPredicate<Person> implements
Predicate<Person> {
```

```
   public boolean test(Person p) {
      return p.getAge() >= 16;
}  }

public class ContactTest {
   public static void main(String[] args) {
      List<Person> people = // get a list of instances
      PersonContact contact = new PersonContact();
      contact.callPeople(people, new DriverPredicate());
}  }
```

- Although this would work we would need a separate class for the definition of all the criteria that might be used

17.13 Implement as AIC

- For more “on demand” criteria, since there might be lots of combinations, we could use an anonymous inner class for the implementation

```
public class ContactTest {
   public static void main(String[] args) {
      List<Person> people = // get a list of instances
      PersonContact contact = new PersonContact();
      contact.callPeople(people, new Predicate<Person>(){
         public boolean test(Person p) {
            return p.getAge() >= 16;
         }  });
}  }
```

- This syntax of the AIC is complex though
  - What was that earlier about what could simplify the syntax of an AIC?...

17.14 Use Lambda Expressions

- With a Lambda expression we can still define the criteria as needed but use a much simpler syntax

```
public class ContactTest {
   public static void main(String[] args) {
      List<Person> people = // get a list of instances
```

```
      PersonContact contact = new PersonContact();
      contact.callPeople(people, p -> p.getAge() >= 16);
}  }
```

- This accomplishes our goals:
  - The criteria is defined “on demand” from the code which needs to indicate what group an action should be applied to
  - The PersonContact class only needs to define methods for the actions, like 'callPeople', 'emailPeople' and 'mailPeople'
- These are really the same benefits of an anonymous inner class but with simpler syntax

17.15 Reuse Lambda Expressions

- Since it is possible that the criteria that are used might need to be reusable, we can do this by having a variable that refers to a Lambda expression
  - This is better than needing to define an entire separate class

```
public class ContactTest {
   public static void main(String[] args) {
      List<Person> people = // get a list of instances
      PersonContact contact = new PersonContact();
      Predicate<Person> drivers =  p -> p.getAge() >= 16;
      contact.callPeople(people, drivers);
      contact.emailPeople(people, drivers);
}  }
```

17.16 Summary

- Lambda expressions are a way to provide implementation of a “functional interface”
- Lambda expressions are a much simpler syntax than anonymous inner classes

Chapter 18 - Input and Output

Objectives

After completing this unit, you will be able to:

- Understand the main components involved in Java I/O
- Understand the benefits of various approaches to working with I/O
- See examples of different methods of reading and writing from a file

18.1 Overview of Java Input/Output

- Java input and output is defined in two main packages
  - java.io - Original part of Java
  - java.nio - "new I/O" added in Java 1.4
  - java.nio.file – "NIO.2" added in Java SE 7 for more advanced file system operations
- There are several components of Java I/O
  - Files
  - Serialization
  - Streams
  - Readers and Writers
  - Buffers
  - Channels

Overview of Java Input/Output

It would be impossible for this lecture to cover everything about Java I/O as there are several dozen classes and interfaces in the various packages. Instead this lecture will provide an overview of the various Java I/O mechanisms and a few examples of reading and writing from files. For more complete documentation refer to the Java API reference.

18.2 The File Class

- Used to obtain or manipulate the information associated with a file or a directory
  - Create or delete a file
  - Check if a File object is a directory or regular file
  - Can check if the file exists, can be read, can be written, is hidden, and the file length
- A File object simply **represents** a file
  - To actually read or write the contents of a file you use the FileInoutStream or FileOutputStream classes

The File Class

The constructors for the File class take various parameters that represent the path to the file. Always remember that this will be interpreted differently depending on the operating system. Operating systems vary in how they separate directories and how they identify the root of the file system.

18.3 File Example

```
try {
String filename = "C:\\sample\\data.txt";
File test = new File(filename);
File parent = test.getParentFile();
if (!parent.exists()) {
       parent.mkdirs(); }
if (test.exists()) {
       test.delete(); }
test.createNewFile();
catch (IOException ioe) {
System.out.println("Trouble with file");
}
```

File Example

The sample code above:

- Checks if the directory exists and creates the entire directory structure if needed.
- Deletes the file if it already exists. If you want to append data to an existing file you would not perform this step.
- Creates a new, empty file. This method will not create a new file if the file already exists which is why the file was deleted in the previous step if we always want a new file.

18.4 Java 7 - The java.nio.file.Path Interface

- Java 7 added the '**Path**' interface as a programmatic representation of a path in the file system as part of the NIO.2 features
  - It can refer to a directory or file
  - Path is an interface because the NIO.2 file system features can be extended so a custom file system may have different ways to implement a Path representation
- The Path interface includes various methods that can be used to obtain information about the path, access elements of the path, convert the path to other forms, or extract portions of a path
- A Path instance will have one or more 'name' elements which are Strings and separated by the path separator of the OS
  - A Path can be an absolute Path if it contains a "root" element (like 'C:\' on Windows and '/' on other operating systems)
  - A Path may be intended to refer to a file or directory depending on the value of the last name element

Java 7 - The java.nio.file.Path Interface

Many of the deficiencies of the previous way to access files was detailed in the NIO.2 Java specification request (JSR 203):

"The Java platform has long needed a filesystem interface better than the java.io.File class. That class does not handle filenames in a way that works consistently across platforms, it does not support efficient file-attribute access, it does not allow sophisticated applications to take advantage of filesystem-specific features (for example, symbolic links) when available, and many of its methods simply return false on error instead of throwing an informative exception."

18.5 Serialization

- In order to be written to a stream, objects must be "serializable"
  - This means the state of the object can be restored in the future
- Classes indicate they are serializable by implementing the **java.io.Serializable** interface
  - This interface is just a "marker" interface as it requires no additional methods

```
public class Customer implements java.io.Serializable
```

  - A class is also serializable if it inherits from a class that is marked as serializable

18.6 Serializing Object State

- When an object is serialized, all of the state is saved
  - This is easy if all the instance variables are primitive types
  - This is more difficult if the object holds instance variables that refer to other objects
- During serialization the entire "object graph" is serialized
  - This means that every object in the hierarchy must also be serializable
  - If any object is not serializable the entire serializing operation will fail
    - A NotSerializableException will be thrown
- Deserialization involves restoring the object state from the serialized information
  - This creates a new object with the same state

Serializing Object State

The "object graph" referred to above is every object that is accessible by starting at the object being serialized and following any references to other objects that are instance variables.

Static variables are not serialized since they are not associated with any particular instance.

18.7 Avoiding Serialization Problems

- Not all classes are serializable
  - Mark these instance variables 'transient' to exclude them from serialization

```
transient NotSerializable notSer;
```

  - Make sure your code behaves properly if these variables are not initialized

```
if (notSer == null) //initialize
```

  - You can also implement the private 'writeObject' and 'readObject' methods to provide your own special handling when an object is serialized
    - This could involve extra steps to remember and restore the state of transient objects that can't be serialized on their own

Avoiding Serialization Problems

Implementing the 'writeObject' and 'readObject' methods is not required in order for a class to be Serializable.

The complete signatures of the methods to override to control object serialization are:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException
```

18.8 serialVersionUID

- When restoring a serialized object, Java needs a way to compare the class definition used to serialize the object with the current definition of the class
  - It does this by looking at the 'serialVersionUID'
- There are two ways to handle this
  - Let Java automatically set this
    - If anything about your class has changed since the object was

serialized the deserialization will fail

- Set your own serialVersionUID
  - This is a static final long class variable

```
static final long serialVersionUID = 38496882344L;
```

  - Make sure you can handle what happens if a previously serialized version of the class is restored
  - Change the serialVersionUID to indicate changes incompatible with previously serialized objects

serialVersionUID

Generally, direct object serialization is not the best way to preserve state over long periods of time. This is especially true if the application is released with several different versions throughout the lifetime of the application. If you want a way to save information over a longer period of time that is more easily adaptable to changes in the information saved it is generally advisable to come up with some type of file saving mechanism that does not rely on object serialization. Ways to do this will be shown later in this lecture. Another advantage of this is you can create a special program for converting files saved in a previous format to be compatible with the new format. With object serialization you would not easily be able to do this.

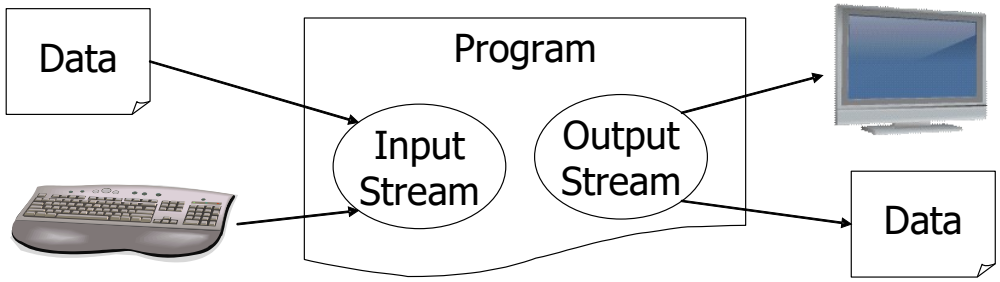
18.9 Options for File Input/Output

- There are several options to write information to files:
  - Direct use of streams
    - Uses streams directly and object serialization
    - Useful for simple situations but files not likely compatible with future versions of the program
  - Readers and Writers
    - Classes available in the java.io package
    - Easier to use than streams but not as much function or performance as Buffers and Channels
  - Buffers and Channels
    - Part of the newer java.nio package

- Provide advanced functionality and improved performance for intricate input/output operations

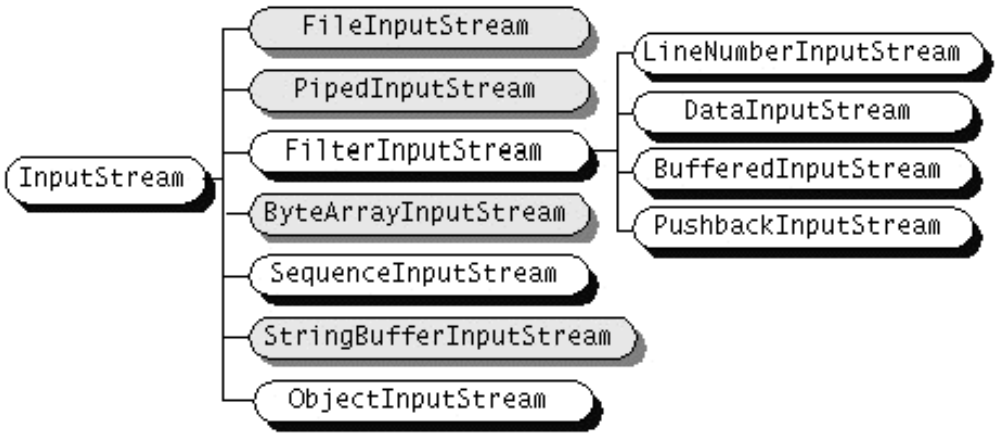
18.10 Streams

- Streams are an abstract representation of an input or output device
- Streams provide common ways to work with input or out no matter what the source or destination is



18.11 Input Stream

- An abstract class
- Provides a source of bytes that your program may read
- All the methods throw IOException



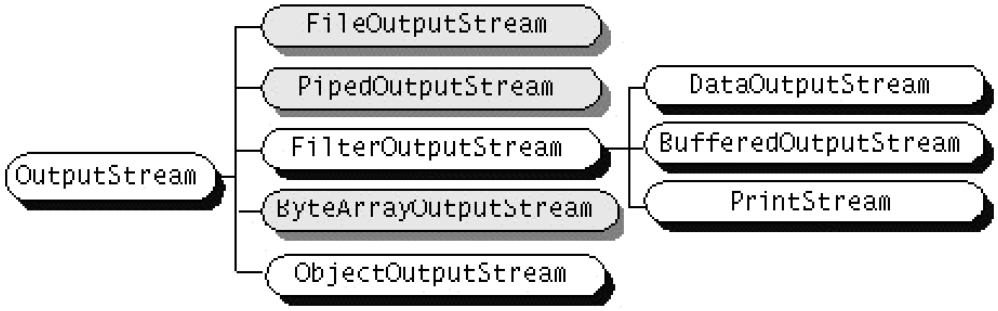
Input Stream

The classes shown above are not all of the subclasses of InputStream but just the ones from the java.io package. There are some other subclasses from other packages for specific purposes.

All of these classes inherit from the abstract InputStream class. The FilterInputStream, for example, overrides the methods of the standard InputStream, and processes the original InputStream in different ways. The DataInputStream further overrides the FilterInputStream by allowing primitives to be read "directly" from the input stream. The abstract method read() is inherited and must be implemented by all subclasses of InputStream.

18.12 Output Stream

- An abstract class
- Provides a destination to which your program may write one or more bytes
- All methods throw IOException



Output Stream

The classes shown above are not all of the subclasses of OutputStream but just the ones from the java.io package. There are some other subclasses from other packages for specific purposes.

Similarly to the InputStream, the OutputStream class is abstract, and is implemented by these classes. By selecting the OutputStream class within the java.io package, and looking at its hierarchy, you can see the classes that use, implement, and override its methods. These subclasses create output streams with different styles.

18.13 "Chained" Streams

- Most streams have a very specific purpose

- It is unlikely that there will be one type of stream that does everything that you need

- In this situation you can "chain" streams together
  - You wrap one stream around another to combine the functionality

```
FileOutputStream fs = new FileOutputStream("file.txt");
ObjectOutputStream os = new ObjectOutputStream(fs);
// Use the combination of streams
```

18.14 RandomAccessFile

- The file streams only allow you to read or to write to a file
  - You are using an "input" stream or an "output" stream
- The java.io.RandomAccessFile class allows you to do both
  - The file is still a sequence of bytes like streams
- You can also "skip" portions of the file instead of reading in sequence
- When constructing a RandomAccessFile you supply the file information and the "mode" the file will be used in

```
RandomAccessFile file = new RandomAccessFile("file.txt",
"rw");
file.skipBytes(48);
double d = file.readDouble();
// Exception handling code omitted
```

RandomAccessFile

In the example above the file is opened in "read/write" mode. All possible modes are:

"r" Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.

"rw" Open for reading and writing. If the file does not already exist then an attempt will be made to create it.

"rws" Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.

"rwd" Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.

18.15 Java 7 – try-with-resources Statement

- Input and Output code will open some resources for the execution of the code and then closes these resources in a 'finally' block
  - Although this is fairly boilerplate code, implementing this pattern can be tricky and omitting a step can cause issues

```
BufferedReader br = new BufferedReader(new
FileReader(path));
try {
  return br.readLine();
} finally {
  if (br != null) br.close();
}
```

- Java 7 adds the 'try-with-resources' statement where resources can be opened within a set of parenthesis on the 'try' block and these will automatically be closed at the conclusion of the 'try' block, making the code easier to write and less error-prone

```
try (BufferedReader br = new BufferedReader(new
FileReader(path))) {
  return br.readLine();
}
```

- Multiple statements can initialize resources if separated by a ';' within the parenthesis

```
try ( ZipFile zf = new ZipFile(zipFileName);
      FileWriter writer =  new FileWriter("foo.out") ) {
```

Java 7 – try-with-resources Statement

Any resource that implements the new 'java.lang.AutoCloseable' interface. This includes all objects that also implement the 'java.io.Closeable' interface.

Note that even though this statement is called 'try-with-resources' the only difference is the set of parenthesis after the 'try' keyword and before the curly brackets for the block of code.

Of all the language changes introduced in Java 7, the 'try-with-resources' statement has perhaps the greatest potential to improve the overall quality of code as developers will no longer need to add the

complex code required to make sure the resources are closed properly.

18.16 Using Streams - Write Example

```
Collection<Customer> custList = new ArrayList<>();
// Fill the customer list
File file = new File("stream.dat");

try (ObjectOutputStream out = new ObjectOutputStream(
    new BufferedOutputStream(new FileOutputStream(file))))
{
       for( Customer customer : custList ) {
              out.writeInt(0);
              out.writeObject(customer);
       }
       out.writeInt(-1);    // Marks end of file
} catch (Exception e) {
       throw new Exception("Trouble with file output", e);
}
```

Using Streams - Write Example

The details of how the 'custList' variable is initialized are not provided because it is not important for this example. The main point being demonstrated is that an object exists that is being written to a file.

The file name can be anything but when using serialization it is good to avoid standard file extensions for text editor or word processing programs.

Writing a '-1' above is just a way of marking the end of the file and is not needed in every situation. This will provide a way to indicate when there are no more objects to be read. A '0' is written before every object to indicate there is an object to read.

18.17 Using Streams - Read Example

```
Collection<Customer> custList = new ArrayList<>();
File file = new File("stream.dat");

try (ObjectInputStream in = new ObjectInputStream(
     new BufferedInputStream(new FileInputStream(file))))
```

```
{
        int custNum = in.readInt();
        while(custNum != -1) {
                custList.add((Customer)in.readObject());
                custNum = in.readInt();
        }
} catch (Exception e) {
        throw new Exception("Trouble reading file", e);
}
```

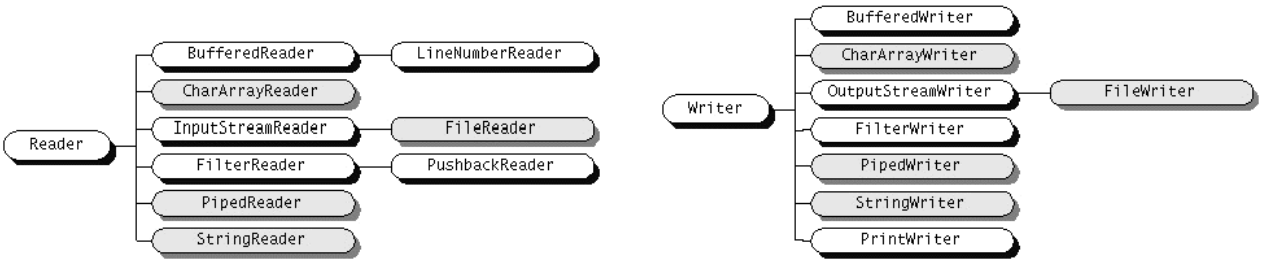
Using Streams - Read Example

In reality you would also want to check the existence of the file separately. This would have been caught because an Exception would be thrown for trying to read from a non-existent file but it is better to check for this first.

In this read example we are using the '-1' written to the end of the file in the previous example to indicate there are no more objects to be read. You can also catch an 'EOFException' which is thrown at the end of the file but be sure no important code is skipped if you use this Exception to indicate the end of the file.

18.18 Reader and Writer

- Part of the java.io package
- Are abstract classes
- Provide partial implementation for i/o
  - Subclasses complete the specifics
- Easier to use that streams but don't have the performance or function of Buffers and Channels
  - Unlike streams that work more with raw bytes and serialization Readers and Writers work with encoded characters allowing for custom file format that might easily be parsed by a future version of the program or even be human readable



Reader and Writer

What if we add an extra field to a Customer class where customer data has been written to a file? Any programs that rely on serialization will likely not be compatible with a file written by a previous version of the program. If we define a custom file format using Readers and Writers we should be able to write a program that will "convert" the file written with the previous program into the file format expected by the current version. We could simply add a blank line or a default value to the file written in the new format for the data of the new field that was added.

Since the Reader and Writer classes are themselves abstract, they implement the core requirements for reading and writing streams of characters, their subclasses implement the remaining requirements. For dealing with characters, and Strings (arrays of characters), Reader/Writer classes come in handy. Instead of using the writeBytes(String) or writeChars() methods of the DataOutputStream class, and converting from characters to bytes and back, a buffer of characters can be written or read at a time. A buffer of characters may be a very long String, and include “carriage return” and “new line” characters.

18.19 Using Readers and Writers - Write Example

```
Collection<Customer> custList = new ArrayList<>();
// Fill the customer list
File file = new File("reader.txt");

try (PrintWriter out = new PrintWriter(file))
{
       for (Customer customer : custList) {
              out.println(customer.getFirstName());
              out.println(customer.getLastName());
              out.println(customer.getCustID());
       }
} catch (Exception e) {
       throw new Exception("Trouble with file output", e);
}
```

Using Readers and Writers - Write Example

When not using object serialization you must decide in what order information will be written out. You would need to match this exactly when reading data back in.

One advantage of not using object serialization is you could write a program to convert the files written by a previous version of the program into the new format expected by the current version of the program.

18.20 Using Readers and Writers - Read Example

```
Collection<Customer> custList = new ArrayList<>();
File file = new File("reader.txt");

try (LineNumberReader in = new LineNumberReader(
       new FileReader(file)))
{
       while (in.ready()) {
              Customer cust = new Customer();
              cust.setFirstName(in.readLine());
              cust.setLastName(in.readLine());
              String idString = in.readLine();
              cust.setCustID(Integer.parseInt(idString));
              custList.add(cust);
       }
} catch (Exception e) {
       throw new Exception("Trouble with file input", e);
}
```

Using Readers and Writers - Read Example

One of the difficulties of reading data in this way is you often have to convert any numeric data. In this example the customer id needed to be converted from a String.

18.21 Using Readers and Writers - Scanner Read Example

```
Collection<Customer> custList = new ArrayList<>();
File file = new File("reader.txt");

try (Scanner in = new Scanner(file))
```

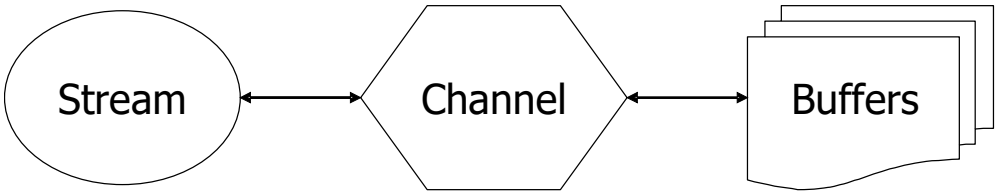
```
{
        while (in.hasNext()) {
                Customer cust = new Customer();
                cust.setFirstName(in.next());
                cust.setLastName(in.next());
                cust.setCustID(in.nextInt());
                custList.add(cust);
        }
} catch (Exception e) {
        throw new Exception("Trouble with file input", e);
}
```

Using Readers and Writers - Scanner Read Example

The previous example is simpler with the java.util.Scanner class. A Scanner can be created directly from a file. The Scanner class also has methods to automatically return the value read as a primitive type.

18.22 NIO Channels and Buffers

- Java NIO is "non-blocking I/O"
  - This was introduced in Java JDK 1.4
- This API is primarily used in "multi-threaded" applications and allow fewer application threads to work with several I/O sources
  - This is common for a server with many open connections
- Java NIO is based around Channels and Buffers
  - Buffers hold the data to be worked with by the application
  - Channels can read data from streams into buffers or write data from buffers into streams
- Using Channels and buffers for large I/O operations can be more efficient than the "standard" I/O process of working with a few bytes at a time
  - Processing data in buffers can be more complex though as you are working with large "chunks" of data at a time.



NIO Channels and Buffers

Working with Java NIO is an advanced topic and details are not provided here beyond this basic introduction.

18.23 Summary

- Java uses streams as the basis for input and output
- Instances of the File class represent files on the system
- Object serialization can be used to save and restore object state
- Readers and Writers provide more advanced functionality than streams

Chapter 19 - Other Java Concepts

Objectives

After completing this unit, you will be able to:

- Describe other Java concepts like annotations, enumerated types, garbage collection, and assertions

19.1 Annotations

- Annotations were a major feature added in Java SE 5
- Annotations provide additional information about Java classes for:
  - Development tools
  - Java compilers
  - Runtime processing
- Annotations can target Java elements like classes, fields, methods, etc
- Annotations are indicated with a '@' and can have name/value pair properties in parenthesis

```
@SuppressWarnings({"serial","unused"})
class MyClass extends AnotherClass {
       @Override
       void myMethod() { .. }
}
```

Annotations

The example above are some of the basic annotations available with Java. If the method that is annotated with @Override does not override a method in a superclass a compiler will generate an error message. This helps avoid a situation where a method that is thought to override a method does not because of a typo or mismatch in parameters. @SuppressWarnings tells the compiler to ignore certain types of warnings when compiling code.

19.2 Enumerated Types

- An "enum type" is a type whose fields consist of a fixed set of constants

- Declared with the **enum** keyword
- Enumerated types are classes that can have methods or other fields
- Often enums are used to declare constants that have different values but are related to each other

```
public enum Gender {
       MALE, FEMALE
}
```

Enumerated Types

Java will also automatically add a values() method that will return an array with all of the values of the enum in the order they are declared.

A more complex enumeration example, with constructors and methods is below.

```
public enum Planet {
         MERCURY (3.303e+23, 2.4397e6),
         EARTH   (5.976e+24, 6.37814e6);  // other planets
         private final double mass;   // in kilograms
         private final double radius; // in meters
         Planet(double mass, double radius) {
                  this.mass = mass;
                  this.radius = radius;  }
         private double mass()   { return mass; }
         private double radius() { return radius; }
         public static final double G = 6.67300E-11;
         public double surfaceGravity() {
                  return G * mass / (radius * radius);  }
         public double surfaceWeight(double otherMass) {
                  return otherMass * surfaceGravity();  }
}
```

19.3 Enumerated Types – Example

- The following code uses the Gender enum declared on the last slide

```
public class Person {
   private Gender sex;
   private String name;

   public Person(String name, Gender sex) {
      this.sex = sex;
      this.name = name;
   }
```

```
   public static void main(String[] args) {
      Person p1 = new Person("Matt Silver", Gender.MALE);
      Person p2 = new Person("Lisa Jones", Gender.FEMALE);
      System.out.println(p1);
      System.out.println(p2);
   }
   public String toString() {
      return this.getName() + ", " + this.getSex();
   }
   // getter and setter methods
}
```

Enumerated Types – Example

A static import can also be used so that references can just use the enum value like 'MALE' instead of 'Gender.MALE'.

```
import static com.mycom.enums.Gender.*;

....
      Person p1 = new Person("Matt Silver", MALE);
      Person p2 = new Person("Lisa Jones", FEMALE);
```

The following code uses the Planet enum declared from the last slide notes

```
public static void main(String[] args) {
         double earthWeight = 175.0;
         double mass = earthWeight / EARTH.surfaceGravity();
         for (Planet p : Planet.values())
              System.out.printf("Your weight on %s is %f%n",
                p, p.surfaceWeight(mass));
}
```

Outputs:

```
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

19.4 Assertions

- An **assertion** is a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed

- Not a replacement for exception handling, rather augment them
  - Assertions can be disabled at runtime
- Add “sanity checks” to your code
  - Assertions can validate code is being used correctly by verifying certain conditions the programmer expects to be true and are part of the documented use of the code
- Handle errors you don’t expect to happen

Assertions

Assertions are boolean expressions that the programmer believes to be true concerning the state of a computer program. For example, after sorting a list, the programmer might assert that the list is in ascending order. Evaluating assertions at runtime to confirm their validity is one of the most powerful tools for improving code quality, as it quickly uncovers the programmer's misconceptions concerning a program's behavior.

As an example of an assertion, let's say a 'deposit' method on an Account class was documented that the parameter passed in was expected to be a non-negative number. An assertion in the code could check this so that the implementation knows that if the assertion passes the value will always be non-negative. An assertion violation in this case means the part of the program that is calling the 'deposit' method is not validating input as it should according to the documented use of the method.

Assertions are not meant to replace the exception handling features in Java. Exceptions handle abnormal conditions arising in the course of the program; however, they do not guarantee smooth or correct execution of the program. Assertions help state scenarios that ensure the program is running smoothly. Assertions should not be used at any place to ensure the smooth running of a program.

19.5 Assertions

- The Assertion Statement can take two forms:
  - assert *Expression 1*;
    - assert 0 < value;
    - assert ref != null;
  - assert *Expression1 : Expression2;*
    - assert ((i/2*23-12)>0) : checkArgumentValue();
    - assert isParameterValid() : throw IllegalParameterError();

        - assert age < 0 : "The value of age cannot be negative" + age;
    - If an assertion fails, AssertionError is thrown
        - The assertion statement with a second expression would be the message supplied with the AssertionError

Assertions

assert Expression1 - The expression is the one the programmer wishes to assert as true. If the expression evaluates to be false, the assertion fails and the system throws an AssertionError error.

assert Expression 1 : Expression 2 - The first argument takes a Boolean expression and serves as assertion condition. The second expression should evaluate to a value which is passed to the AssertionError constructor. This option allows the programmer to provide more detailed information about the problem. The second expression cannot be a method call that returns void.

The assertion statement is a shorthand way of checking a condition and throwing an AssertionError if it is violated. Some of the examples from the slide are equivalent to:

```
if(!(ref!=null)) {
  throw new AssertionError();
}

if(!(age > 0)) {
  throw new AssertionError("The value of age cannot be negative" + age);
}
```

19.6 When to use Assertions

- Situations when assertions should be used:
    - Internal invariants – assumptions concerning a program's behavior
    - Control-flow invariants – assumptions about the flow of control
    - Preconditions of non-public methods – conditions that must always be true just prior to the execution of some section of code or method
    - Postconditions – conditions that must always be true after the execution of some section of code or method
- There are some cases where assertions should NOT be used:
    - Checking arguments in public methods – Handle incorrect method parameter values with Exceptions since those can't be disabled
    - Use assertions to do any work required for correct operation – If

assertions are disabled this work won't be performed

- Rather than just assuming certain conditions are met, assertions can enforce these conditions and allow the code to have documentation of these expectations

When to use Assertions

Ensure that a program works in a pre-determined manner: Assertions can be used within programs to make sure the program behaves in a predetermined manner and will throw an error when violated. For instance, an assertion can be placed in the code below to declare that interest will never be negative.

```
if (interest > 1000)  {
          principal = principal + 100;
}
else if (interest < 1000 )  {
          assert interest > 0 : "Interest cannot be negative";
          principal = principal + 10;
}
```

Pre-Conditions: Assert statements can be used to check the validity of the parameters passed before they get used in the body of the method. However, java warns that assert statements should not be used to check parameters of a public method.

```
private int checkInterest (int interest)  {
          assert interest > 0 : "Interest cannot be negative."
          // Do computation with interest.
}
```

Post-Conditions: For instance, assert statements can be used to check for the validity of the returned values in a method that has multiple return statements.

19.7 Assertions Examples

```
private int checkInterest (int interest)  {
        assert interest > 0 : "Interest cannot be negative."
        // Do computation with interest.
}

if (i % 3 == 0) {
        ...
} else if (i % 3 == 1) {
        ...
} else {
```

```
        assert i % 3 == 2 : i;  // true if other tests false
        ...
}
```

19.8 Enabling Assertions

- By default, assertion feature is disabled
- To enable assertion during runtime, use these command line options:

```
-ea   // Enable assertion for all classes

-ea:com.webage.utils...
  // Enable assertion for a package (note the three dots)

-ea:com.webage.utils.ZipSearch
  // Enable assertion for a class only
```

- Often assertions might be enabled in development and testing and then disabled in production to avoid the overhead of evaluating all of the assertion expressions
  - You could also enable assertions in production temporarily if needed to help track down where a program violates conditions for difficult bugs

Enabling Assertions

The assumption over enabling assertions in development and testing and then disabling them in production is that all of the assertion violations that would have caught code that is being used incorrectly have already been caught and corrected. If we have extensively tested the code that calls the 'checkInterest' method of a previous example and found that it never calls the method with a negative value (even when we try to force it to) the deposit method no longer has to check this assertion. Disabling assertions is probably best when you have a robust testing capability that routinely tries to force the program into error conditions but the program handles it "properly" before generating assertion failures.

“ea” stands for **enableassertions**.

The option –**da** can be used to disable assertions for all classes or selected classes and/or packages.

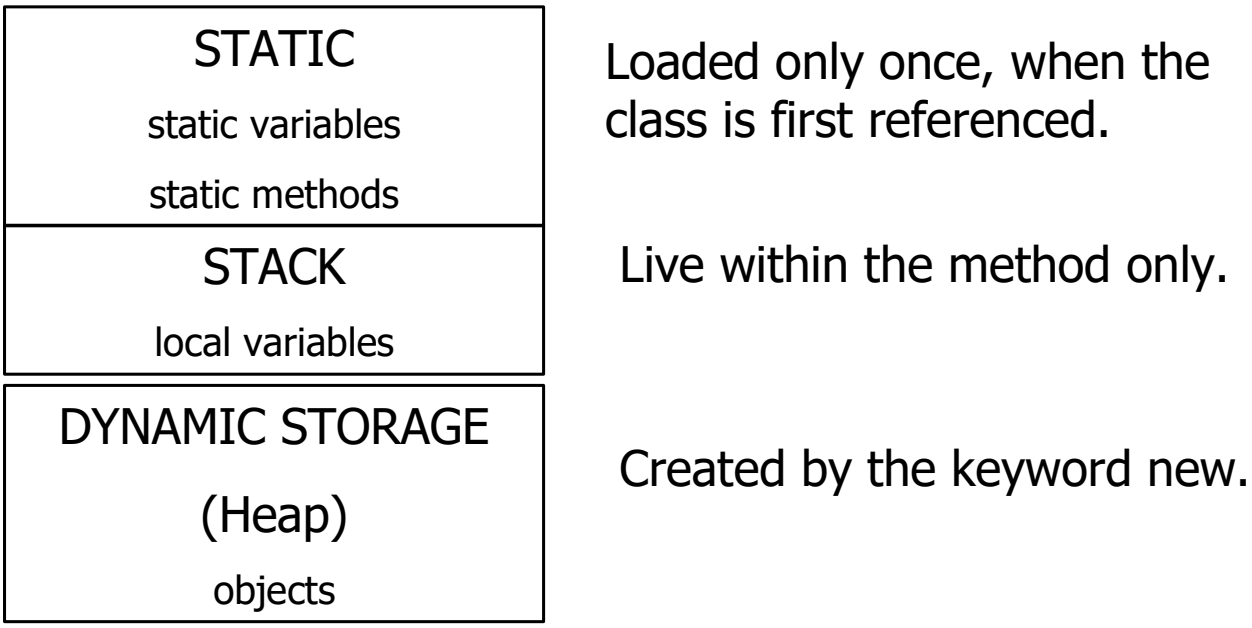
Example to enable assertion for all classes and packages except for pkg1.

```
java –ea –da:pkg1… AssertPackageTest
```

In order for the javac compiler to accept code containing assertions, you must use the -source 1.4 command-line option as in this example:

```
javac -source 1.4 MyClass.java
```

19.9 JVM Storage Areas



JVM Storage Areas

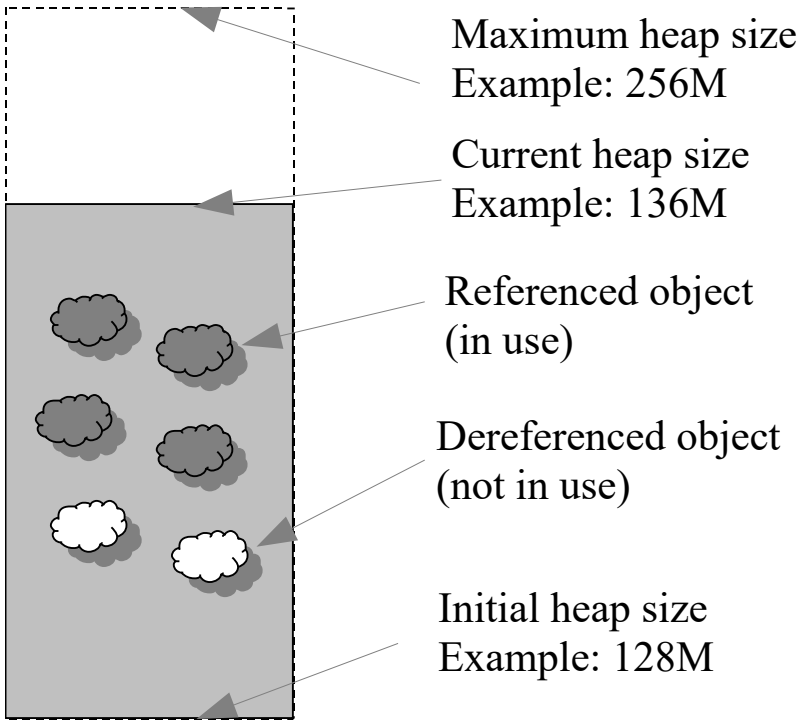
Static - Loaded only once, when a class is first instantiated or when a static variable or method is first **referenced**. When objects of the class are garbage collected, the static items are not unloaded. The life of the static items is typically the life of the Virtual Machine.

Stack - Local variables are allocated in the stack; it is not initialized by the system. The compiler ensures that the method code initializes them (or you will get a compiler error). When the method is entered, memory is allocated and the area is deallocated (ceases to exist) when it exits the method.

Dynamic Storage - all instantiated objects.

19.10 Java Heap Space

- Heap is a reserve of memory maintained by the JVM.
- The JVM allocate memory from the heap for creation of objects.



Java Heap Space

Every object created by a Java application requires memory. JVM allocates this memory from a massive reserve of memory called the **heap**. Memory for the heap itself is obtained by the JVM from the underlying operating system. When the JVM starts up, certain amount of memory is allocated to create the heap. This is called the **initial heap size**. The JVM allocates more memory from the OS to grow the heap if needed. The heap can only grow to a predetermined limit. This is called the **heap maximum**. Portion of the heap that is not currently allocated to create objects is called **free memory**. The size of the heap may also shrink if the JVM realizes that too much of the heap is going unused. The heap never shrinks below the initial heap size.

It is worth noting that the objects that are no longer in use by the application continue to occupy memory in the heap until they are garbage collected.

OS Level Memory Management

An operating system manages any memory, including that allocated to the JVM for heap, as virtual memory. Virtual memory is divided in regions called pages. A page is usually 4K in size. For example, a 32KB virtual memory segment will have 8 pages. The memory is called "virtual" because a page may reside in fast physical memory (RAM) or slow disk based memory. The exact location of a page is hidden from the application.

The OS will try its best to allocate a page in the physical memory so that the application can perform well. If no physical memory is available, a page is allocated on disk. When the application accesses memory in a page that is on disk, the OS tries to load the memory into physical memory. This operation is called **page in**. If it finds free space in RAM, this operation succeeds. Otherwise, the OS looks for a page in RAM that have not been used in a while. Such a page is then copied into disk. This is called **page out**. The memory for that page is freed up to make room for the page that is being accessed by the application. The page that just got paged out is called the **Least Recently Used (LRU) victim**. Ideally, when you have sufficient RAM available, there is no need for page in or page out. When the machine is running out of physical memory, pages are frequently paged out and in.

Paging in and out are collectively and popularly known as simply **paging**.

Copying data from disk to RAM and vice versa is an expensive operation. When an application accesses a page that needs to be paged in, the application is halted until the page in operation is complete. Aside from slowing down the application, paging in and out puts heavy demand on the disk controller. In a machine with high page in and out rate the disk I/O level is also very high. This puts a double whammy on a disk driven application such as a database.

19.11 Heap Size Limits

- Size of heap can grow or shrink. JVM allocates as much heap as needed to create the objects.
- The size can not be less than the initial heap size or more than the maximum heap size.
- The limits are configured through command line options:

```
java.exe   -Xms128m   -Xmx1024m   MyClass
```

- -Xms has a very definite impact on the heap size behavior where as -Xmx is a ceiling only.

Initial Heap Size

When the JVM starts up it allocates a certain amount of memory for the heap. This is called the initial heap size. The size of the heap expands or shrink as needed. The size can never exceed the maximum heap size or be less than the initial heap size.

The value of the initial heap size is set using the -Xms command line argument. For example, -Xms128m will allocate 128MB initial memory.

Maximum Heap Size

The largest possible size of the heap is called maximum heap size. This is set using the -Xmx command line parameter. For example, -Xmx256m will set the maximum limit to 256MB.

You can not set an initial heap size that is higher than the maximum heap size.

As the application uses more and more objects JVM needs to expand the size of the heap. If it reaches the maximum and still more memory is required, an java.lang.OutOfMemoryError exception is thrown. The application can behave very unpredictably at that point.

Current Heap Size

This is the size of the heap at any point in time. This size can not be less than the initial heap size. The maximum heap size has no definitive impact on the current heap size. Where as the initial heap size has a very definitive impact. This is often a source of confusion. Let us say that we start an application as follows:

```
java.exe   -Xms128m   -Xmx1024m   MyClass
```

The heap size at any point of time will be at least 128MB even if the application is really frugal in its memory usage and doesn't really need any more than 10MB memory for all the objects it creates. Setting an aggressively high initial heap size can waste memory unnecessarily.

Now, consider an application that is more memory intensive. It may need 200MB to 250MB memory. The heap will grow to only what is needed. In other words, the maximum memory (-Xmx) is a ceiling only but not in anyway a hint to the JVM on how much memory to actually use.

You can safely set a high maximum size without worrying about using too much memory from the OS. If the application is well behaved and does not have any memory leaks, the heap will only be as large as needed. A high value of maximum size is just a safeguard. If, for some reason, there is a temporary and unexpectedly high demand for memory, the application will not fail. A high maximum limit will give the heap room to grow. Once this unusual condition ends, the heap will automatically shrink to its normal level. Setting an aggressively low maximum limit eliminates this room for any unusual situation.

19.12 Garbage Collection Basics

- In Java, objects are explicitly created by application code but not destroyed.
- Periodically, the JVM goes through all objects that are of no use to the application and frees up the memory allocated to them. This is called garbage collection (GC).
- The freed up memory "goes back" to the heap. Futures objects are

created using this freed up memory.

- While GC is in progress, the application may perform poorly. We will find ways to minimize the time spent in GC.

Garbage Collection

In a Java application when an object is created, memory is allocated for it. But, Java provides no way of explicitly destroying an object. In contrast, programming languages like C and C++ provide a way for the programmers to write code to destroy an object which frees up the memory used by the object. Java automatically destroys objects that are no longer in use by the application. This process is called **garbage collection (GC)**. Here, garbage refers to unnecessary objects that are currently taking up memory from the heap.

When objects are created and when they become unnecessary are questions that the programmers deal with. As an administrator, you have very little control over it. You need to simply know that an object may be still in use by the application or it may not be in use anymore. If it is no longer in use, JVM will eventually return the memory back to the heap during garbage collection. Future objects can then use that freed up memory.

Following is a very simple code example that shows how objects are created and marked as unused.

```
String obj = new String(); //An object is created
obj = null; //Object is no longer in use
```

A slightly more complex example shows how an object can remain in use.

```
String obj1 = new String(); //An object is created
String obj2 = obj1;
obj1 = null; //Object is still in use
```

Here, the program variable obj1 is marked as unused. But, the variable obj2 is still referring to the object. The object will not be destroyed during garbage collection.

JVM performs garbage collection periodically and as needed. An application may function poorly or may not respond at all while the garbage collection is in progress. We will learn a few ways to minimize the time spent in GC.

19.13 Allocation Failure (AF)

- The memory for an object must be a contiguous block of bytes.
- When an object is created, if JVM can not find a block of memory big enough for the object, allocation failure takes place.
- AF leads to GC.

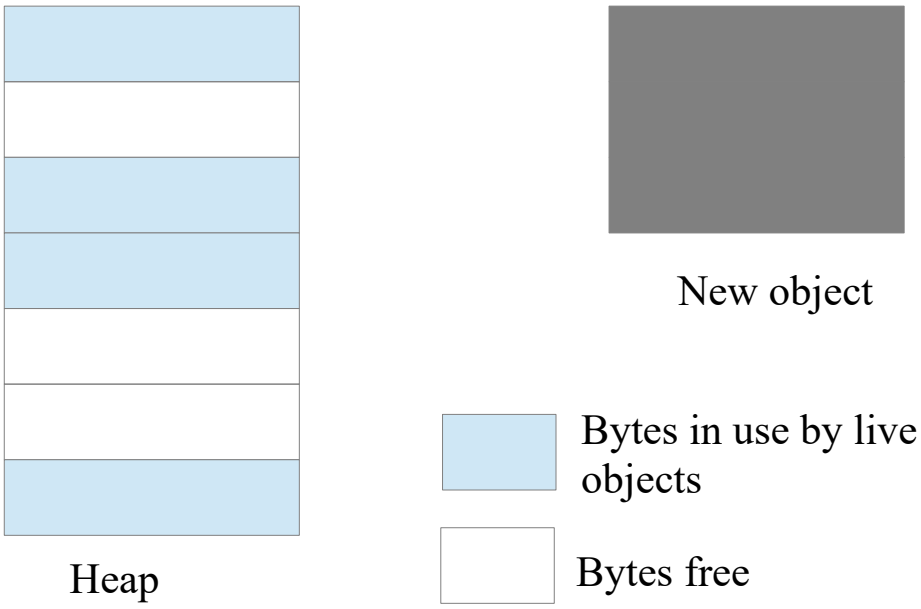


Diagram showing a situation where no contiguous block of memory exists in the heap to fit in a new object.

Allocation Failure (AF)

When the application code creates a new object, JVM looks for a contiguous region of memory in the heap big enough for the object. For example, if an object takes up 64 bytes, there has to be 64 contiguous free bytes in the heap. If JVM can not locate this block of memory, AF takes place. AF automatically leads to garbage collection. After GC, JVM repeats the search for a free block of memory for the object.

Note, AF does not mean there is no free space left in the heap. It simply means, none of the free regions are big enough for the object. If you are monitoring the heap using some kind of a tool, you will observe AF and GC taking place even though there is enough free space available in the heap.

19.14 OutOfMemoryError

- An out of memory error occurs when the JVM needs to create a new object and can't free up enough memory through any type of garbage collection for that object
  - This happens because all of the objects are referenced and the remaining free space is not large enough even if it is all compacted together

- An out of memory error can indicate two things
  - The maximum heap size is set too low for the natural demands of the application
  - The application contains a memory leak

19.15 Memory Leak

- This is a logical error in the application that allows object references to remain for a long period of time even though they are no longer being used by the application
- Garbage collection only collects *unreferenced* objects so an object that is referenced but is not used in the future is not cleaned up but does not provide any value to the application
- The biggest symptom of a memory leak is that the heap never reaches a steady state and gradually grows as the application runs
  - The rate of growth may be different depending on the cause of the leak but there will generally be a continuous upward trend in the used memory
- The ultimate outcome of a memory leak is to cause an OutOfMemoryError
  - Hopefully analysis tools can help you determine if a memory leak is present before this occurs
- A temporary solution to a memory leak is to restart the Java program or process to clear out all objects and let the application create new objects to handle requests
  - The permanent solution is to install a new version of the application that allows objects that will not be used to become unreferenced so they can be garbage collected

Memory Leak

Besides restarting the server there is nothing that can be done administratively to prevent a memory leak. Even giving the application more memory will just extend the amount of time it takes to fill up the memory from the leak. Many tools exist to help evaluate the structure of object references in the JVM heap and help look for possible causes of memory leaks. These "candidates" should be evaluated by developers to determine if there should be an increasing number of objects of that type remaining

referenced in the application.

19.16 Distributing Java Code with JARs

- Since a Java program contains many classes, defined in separate files, distributing the code as separate files would be a challenge
- To solve this problem Java code and other resources can be bundled into a single JAR file (Java ARchive)
  - JARs can be created by development tools build processes
  - JARs are in standard Zip file format and can be examined using most zip file utilities
- A JAR can optionally have a "manifest file" that contains certain settings
  - This is always META-INF/MANIFEST.MF
  - If the packaged code has a class with a 'main' method that runs the program the manifest can list this so the JAR is "executable"
  - The manifest can "seal" packages to ensure that the only source of code from that package is from the same JAR
  - The manifest can declare other code that must be on the "Classpath" in order for the code of the JAR to operate
- Once a JAR file of code is available it may be imported or linked to other projects to use the code

Chapter 20 - Introduction to Gradle

Objectives

Key objectives of this chapter

- What is Gradle
- Why Groovy
- Build Script
- Task Dependencies
- Plugins
- Dependency Management
- Gradle Command-line Arguments

20.1 What is Gradle

- Gradle is a flexible general purpose build tool like ANT
- Provides powerful dependency management
- Full support for your existing Maven or Ivy repository infrastructure
- Ant tasks are also supported
- Groovy and Kotlin languages are supported for writing scripts
- It is free and an open source project, and licensed under the Apache Software License (ASL)
- Gradle can increase productivity, from single project builds to huge enterprise multi-project builds

20.2 Why Groovy?

- ANT uses declarative XML based style
- Gradle uses a DSL (domain specific language) based on Groovy which makes it more powerful
  - Also supports Kotlin

- Gradle's main focus is on Java projects, but it's still a general purpose build tool
- Groovy offers transparency for Java people
- Groovy's base syntax is the same as Java's

20.3 Build Script

- build.gradle file
  - Analogous to ANT's build.xml file
- Written in Groovy
- Composed of tasks
  - Similar to ANT targets
- Tasks are composed of actions
  - doFirst, doLast

20.4 Sample Build Script

```
task hello1 {
   doLast {
      println 'Hello World!'
   }
}

task hello2 {
   doFirst {
      print 'This is '
   }
   doLast {
      println 'a test!'
   }
}
```

20.5 Task Dependencies

- A task can depend on one or more tasks
- If a dependent task already exists earlier in the build script, then the following syntax can be used

```
task <task_name>(dependsOn: dependentTask)
```

- If not, then the dependent task name must be enclosed in quotes like this:

```
task <task_name>(dependsOn: 'dependentTask')
```

20.6 Plugins

- Plugins can be defined in the build script to make more tasks available to Gradle
- Examples
  - Java plugin – provides tasks for compiling Java code, copying resources, creating a JAR file, running unit tests and generating JavaDocs
  - Application plugin – provides a task to create an executable JVM application. Packages application as a TAR and/or ZIP file and includes operating system specific start scripts.
- Add plugins to build.gradle file using the following syntax:

```
plugins {
    id 'java'
    id 'application'
}
```

20.7 Dependency Management

- A project can make use of additional libraries that are not already part of current project
- Java projects can connect to various repositories, such as Maven Central and JCenter
- Repositories can be defined like this:

```
repositories {
```

```
    jcenter()
}
```

- Dependencies can be defined like this:

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

20.8 Gradle Command-Line Arguments

- gradle tasks: displays the tasks defined in the build script, including plugin provided tasks
- gradle -q: suppresses the log messages and runs the default tasks
- gradle build: compiles code, generates JAR file and runs the unit tests
- gradle clean: cleans the build directory
- gradle test: runs the unit tests

20.9 Summary

- Gradle is an open source, general purpose build tool
- Gradle offers more flexibility since it uses Groovy as a DSL language, which is closer to Java, instead of using XML based syntax
- Gradle allows dependency management for packages / libraries

Chapter 21 - Appendix: Overview of Java SE APIs

Objectives
After completing this unit, you will be able to: ■ Understand the function provided by some of the other main packages in Java

21.1 Java GUI Programming

- There are several packages provided to create graphical Java programs
  - **java.applet** - Create graphical programs that run on a web page
  - **java.awt** - Provides mechanism for drawing graphics and images
  - **javax.swing** - Provides a way to create windowed programs with prompts, forms, etc
- Using the Swing API is preferred when possible because it relies less on direct support from the operating system
  - This means applications will have a more consistent look on different platforms

Java GUI Programming

You can learn more about this by using the following Java "tutorial":

http://java.sun.com/docs/books/tutorial/uiswing/

21.2 Networking

- Provided by the **java.net** package
- Provides two types of APIs
  - "Low level" API with IP addresses, sockets, and network interfaces
  - "High level" API with URIs, URLs, and connections
- Some of the classes provide I/O interaction with InputStream and OutputStream
- There is also a **javax.net** package with support for creating sockets and

SSL

21.3 Security

- Provided by the **java.security** and **javax.security** packages
  - Many different capabilities related to security are in these packages
- Contains a robust security model for permissions, policies, access control, authentication, and security certificates
  - You can make decisions in code execution depending on user attributes
- Provides very fine-grained security policies
  - You can indicate that a program can only read/write files from a particular directory, open network connections only to a specific host, etc
  - This provides a central mechanism to protect from malicious code that may have been added
- Security policies are enforced by the java.lang.SecurityManager class
- The **javax.crypto** package also provides cryptographic operations

21.4 Date and Time API

- Java SE 8 added a new Date/Time API in the new '**java.time**' package and subpackages
- The Date-Time API consists of the primary package, java.time, and four subpackages:
  - **java.time** - The core of the API for representing date and time. It includes classes for date, time, date and time combined, time zones, instants, duration, and clocks. These classes are based on the calendar system defined in ISO-8601, and are immutable and thread-safe.
  - **java.time.chrono** - The API for representing calendar systems other than the default ISO-8601. You can also define your own calendar system. This tutorial does not cover this package in any detail.

- **java.time.format** - Classes for formatting and parsing dates and times.
- **java.time.temporal** - Extended API, primarily for framework and library writers, allowing interoperations between the date and time classes, querying, and adjustment.
- **java.time.zone** - Classes that support time zones, offsets from time zones, and time zone rules.

Date and Time API

Prior to the inclusion of the Date/Time API with JSR-310 in Java 8, the open source project Joda-Time provided the same functionality to improve working with date and time in Java. It was the de facto standard date and time library for Java prior to Java 8.

21.5 Databases - JDBC

- The **java.sql** and **javax.sql** packages provide a way to connect and query relational databases
  - This is better known as the JDBC API
- A database vendor can provide a JDBC "driver" to allow Java code to use this mechanism
- JDBC code is written in a standard way that does not rely on one particular database
  - This makes the code very maintainable as different databases could be used over the lifetime of an application
- You can cache or page through data which reduces memory consumption and database locking

21.6 Concurrent Programming

- Java has several ways to support concurrent or parallel processing
- From the beginning Java has had classes like **java.lang.Thread** and the **java.lang.Runnable** interface that can be used to execute code in parallel
  - These can be used for basic parallel programming but can be difficult to use for more advanced concurrent programs

- Java SE 5 added the **java.util.concurrent** pacakge which provides a high level API which is easier to use for advanced uses
  - **Executors** provide implementations that can run code with things like thread pools or on a schedule so that these features do not need to be built from scratch
  - **Locks** that provide more advanced control than the basic synchronization utility built into Java
  - Many classes like concurrent queues and collections so these can be used in a multi-threaded program
- Java SE 7 introduces the "Fork/Join" framework to Java concurrency to take advantage of multiple processors
  - The code to use this would recursively break down a problem into smaller tasks until the task is simple enough to solve

21.7 Collections “Stream API”

- Using collections is very common in Java programs
  - You are usually collecting objects to DO SOMETHING with them
- Aggregate operations are functions that take a group of elements as input and perform some action or calculation
  - Examples would be finding maximum, minimum, average, sum, etc
- Java 8 adds the Collections “Stream API” to provide this kind of functionality
  - There is a new '**java.util.stream**' package although classes and interfaces in the Collections framework were modified also
- There are two concepts central to the Stream API:
  - **Pipelines** – A pipeline is a sequence of aggregate operations
  - **Streams** – A stream is a sequence of elements

Collections “Stream API”

One of the benefits of the Stream API is that by not needing to code the implementation, the platform

can optimize how a aggregate operation is performed. The patterns of implementing some of these operations is so common that it is better if the Java platform “does it for us”.

21.8 Functional Interfaces for Lambda Expressions

- Lambda expressions must implement a “functional interface”
  - This is an interface with a single abstract method
- The **java.util.function** package was added in Java 8 and provides a number of “built-in” functional interfaces that can be used as the basis of a Lambda expression
  - Although not all possible functional signatures that might be needed are provided, the most common patterns are all represented
- The interfaces in the package leverage generic qualifiers heavily to modify the signature of the interface
  - This allows a developer to still control the types involved without needing to define a completely custom interface
- The various places in the Java API can use these built-in functional interfaces to define what kind of Lambda expression needs to be provided
  - For example, the '**Stream.filter**' method takes a parameter of '**Predicate<? super T>**' to indicate that it needs a parameter that can operate on the elements of the Stream and return a boolean to indicate if the element should be part of the Stream that results from the filtering
- You can also call the method of the functional interface “manually” to invoke the Lambda expression that might be a variable of that type

java.util.function Package

Knowing the contents of the 'java.util.function' package will help understand a large part of the API changes in Java 8 and also help more effectively use Lambda expressions.

21.9 Naming - JNDI

- Provided by the **javax.naming** package
- Describes the "Java Naming and Directory Interface" or JNDI

- Provides a way to "bind" objects to a name and then do a "lookup"
  - This is independent of the JNDI implementation
- Makes applications more modular as they can "discover" other components at runtime
- Key feature used by applications deployed to Java Enterprise Edition application servers

21.10 Management - JMX

- Provided by the **javax.management** package
- Defined by JMX or the Java Management Extensions
- Standard API for managing and monitoring Java applications
  - Consult and change application configuration
  - Monitor statistics about application behavior
  - Notification of state changes and errors
- Based on the concept of "MBeans"
  - A Java object that represents a managed resource
- Many commercial Java software products use JMX to provide management and monitoring capability

21.11 XML

- JAXP, or the Java API for XML Processing has many ways to validate, parse, transform, and write XML
  - **org.w3c.dom** - DOM (Document Object Model) processing of XML by representing the document in memory
  - **org.xml.sax** - SAX (Simple API for XML) processing which is event-driven and often better for looking for specific data in large documents
  - **javax.xml.stream** - StAX (Streaming API for XML) which represents a cursor within an XML document allowing the navigation of a document like DOM but without the overhead

- **java.xml.transform** - XSLT transformations on XML documents
- Other packages for datatypes, namespaces, and validation of XML

- JAXB, or Java Architecture for XML Binding provides an easy way to convert the structure of Java classes to XML data
  - This is in the **java.xml.bind** packages
- XML has become a very common way to transmit data or store information like configuration

XML

The StAX API was added to Java SE 6 although there is a separate implementation available for Java SE 5.

21.12 Web Services

- Web service support has been in Java Enterprise for a while
- In Java SE 6 many web service APIs were moved into Java SE
  - This was mainly to allow web service clients to be written with Java SE environments without any additional libraries
- **JAX-WS** is the Java API for XML Web Services
  - **javax.jws** package has web service annotations for use with Java classes
  - **javax.xml.ws** has the core APIs
  - **javax.xml.soap** define classes for SOAP web services which are common

21.13 Remote Method Invocation

- Provided by the **java.rmi** and **javax.rmi** packages
  - Allows code running in one Java JVM to invoke a method on a "remote" JVM
  - Performs "marshalling" and "unmarshalling" which generally involves

serializing objects over a remote connection

- The Java Enterprise Edition provides "Enterprise JavaBeans" (EJB) which are based on RMI although you do not have to directly write RMI code

21.14 Image I/O

- Provided by the **javax.imageio** package
  - Describes content of image files, including providing thumbnails
  - Mechanism for reading and writing images
  - Ability to transcode between formats
- Works with standard formats like JPEG, PNG, BMP, and GIF

21.15 Printing

- Provided by the **javax.print** package
  - Discover and select print services based on capabilities
  - Specify printed data format
  - Submit print jobs

21.16 Summary

- The Java language contains many APIs for various purposes
- Knowing what these APIs are and where to find more information can make application programming much easier

Summary

You can get more information on various Java SE APIs at:

http://docs.oracle.com/javase/

Chapter 22 - Appendix: Overview of Java EE

Objectives

After completing this unit, you will be able to:

- Understand the basic concepts of Java EE
- Describe the role of an "Application Server"
- Describe the major components of a Java EE application

22.1 Goals of Enterprise Applications

- Provide secure access to data from many types of client devices
- Reduce cost of development and maintenance to provide increased ROI
- Allow new features to be added without major redesign of an application
- Integrate existing systems
- Provide an infrastructure that can easily scale to meet increased demand

22.2 What is Java EE?

- Java Platform, Enterprise Edition
- Extends Java SE with additional support for client-server applications
  - An additional “API” specific to Java EE applications
- Provides a common specification of an “Application Server” and the services provided by that Application Server
  - The Application Server is the Java process which is the JVM running deployed Java EE applications
- Defines major application components and APIs:
  - Servlet
  - JSP
  - EJB
  - JSF – Java Server Faces

- CDI – Contexts and Dependency Injection
- JPA – Java Persistence API

- Allows developers to focus on business logic rather than infrastructure
- Provides flexibility in choice of vendors as many offer Java EE compliant software

What is Java EE?

Note: The Java EE specification was previously known as the J2EE specification. The name was officially changed with the release of the Java EE 5 specification. You may still see the term "J2EE" in books, articles, or product documentation.

22.3 The Java EE Specifications

- Define the programming model for component-based Enterprise Applications
  - The Java EE API extends the standard Java API
- Define a standard Application Server environment to execute these components
- Define a standard way for packaging components of an Enterprise Application
  - This includes "Deployment Descriptors" with deployment information about Java EE applications
- Provide for a flexible architecture

22.4 Versions

- There are various versions of the different specifications
  - Knowing how they relate to each other can be important when looking at documentation

Java SE	5	6	7
Java EE	5	6	7
Servlet	2.5	3.0	3.1
JSP	2.1	2.2	2.3
JSF	1.2	2.0	2.2
EJB	3.0	3.1	3.2
JPA	1.0	2.0	2.1
CDI	None	1.0	1.1
SOAP Web Service	JAX-WS 2.0	JAX-WS 2.2	JAX-WS 2.2
REST Web Service	None	JAX-RS 1.1	JAX-RS 2.0

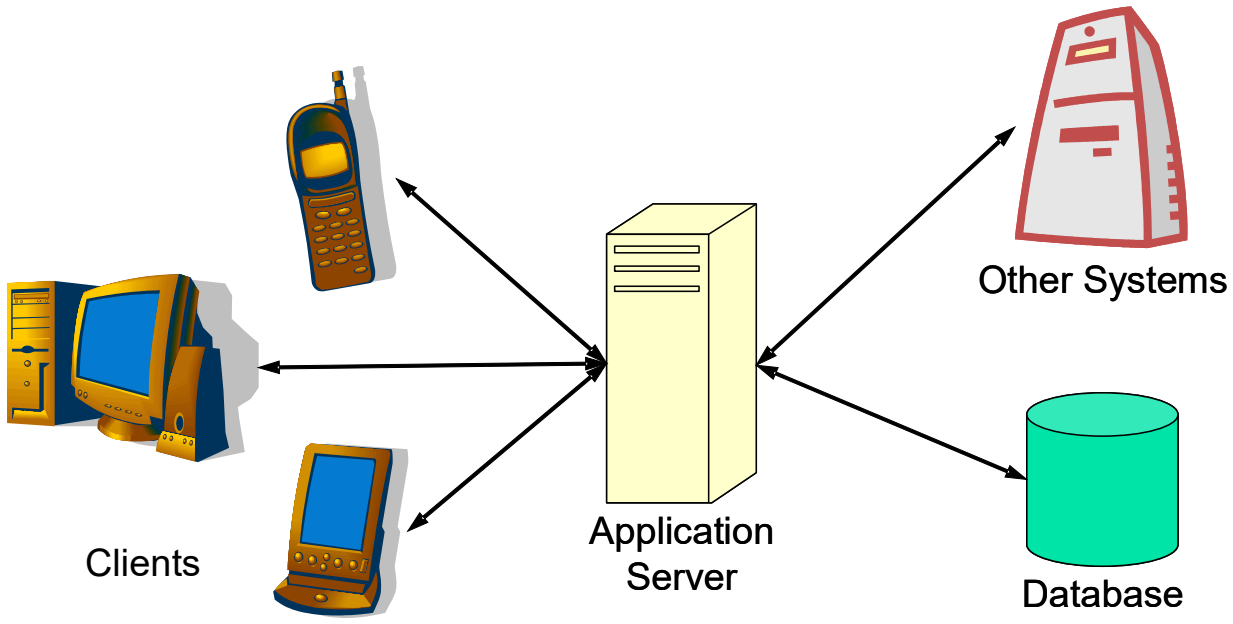
Versions

The version of Java SE listed above is simply the "minimum" level of Java SE required. Many Java EE software products have versions that run more recent versions of Java SE since the Java SE specifications often release a new version before a new version of the Java EE specification is released.

The Java SE and Java EE naming conventions were restructured recently. Previously there was a '2' in the name to indicate "Java 2" and the release numbers were 1.x. This will be confusing until the new naming conventions are the only ones used in books, articles, and documentation.

22.5 Role of Application Server

- “Multi-Tiered” Application
  - Client, Application Server, Data



Role of Application Server

Represents standard n-tier architecture.

The browser serves as a universal client. Typically web pages and JSPs invoked from the browser interact with servlets deployed in the application servers. Application servers access business logic, and objects in the form of EJBs and JavaBeans in the back end of the server.

The application server may access backend systems like databases, SAP etc.

22.6 Java EE Components

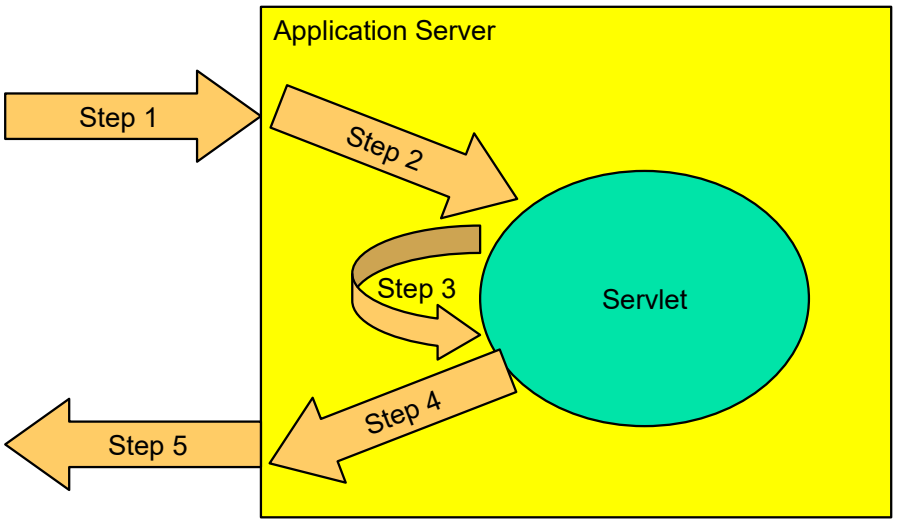
- Server-side Components
  - Servlet
  - JSP
  - EJB
  - Java classes
- Client-side Components
  - Web Browser

- Java Applets
- Java “Application Clients”

22.7 What is a Servlet?

- Java code designed to handle specific client requests
- Coded by developers with a few special operations which will be called by the server
  - These will be called when a client request for the Servlet is received
  - Define what code will execute when this happens
- Good for: making decisions, handling errors, delegating responsibility to other classes
- Not good for: displaying output to a client using HTML, directly manipulating data

22.8 Servlet Execution



Servlet Execution

1. Client submits request to Application Server
2. Application Server uses URL to find Servlet designed to handle the request

3. Servlet executes code written by application developers to process request
4. Servlet delivers response to be forwarded back to client
5. Application Server sends the response back to the client using the protocol that was used to send the request

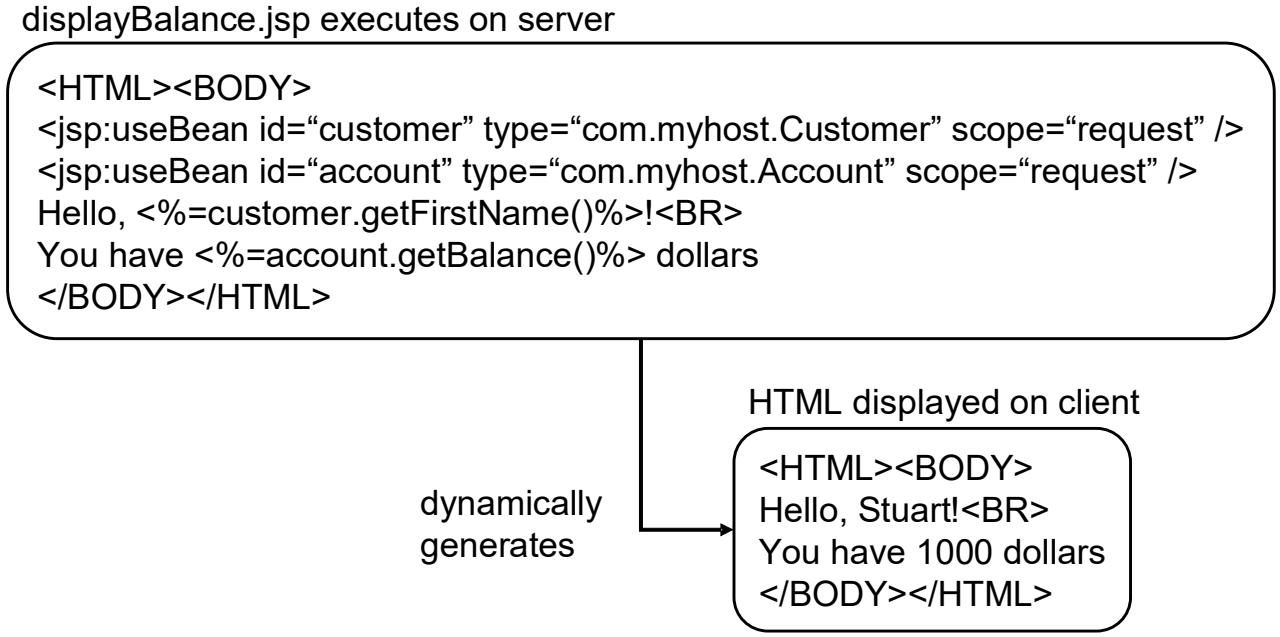
22.9 What is a JSP?

- JavaServer Page
- An HTML file with special “JSP tags” to display dynamic content
- Dynamically processed by the server when a request is received
  - JSP tags only execute on the server
  - Client just sees standard HTML output
- Good for: displaying output to a client using HTML, using HTML editing tools for design
- Not good for: handling complex client requests, directly manipulating data

What is a JSP?

JSP files can use other markup besides HTML. HTML for Web Browsers is most common.

22.10 JSP Code Sample



JSP Code Sample

The code from the JSP file above is the contents of the file as it exists on the server. The special tags access information from other components on the server and display values in the output. Just like a Servlet, the JSP only executes when a request is received from a client. When the output is received by the client, they only see standard HTML which includes the values that were dynamically generated. They may not even know that a JSP was used to generate the output.

22.11 Introduction to JSF

- JSF is JavaServer Faces technology
- A Java EE specification to develop web based applications quickly and easily
- Provides these key functionalities:
  - Ability to define a screen's layout in a web page using JSF custom tags
  - Ability to associate a GUI control (such as text box) with a JavaBean's property for data input and output
  - Ability to handle events such as form submission in a JavaBean

method

- Ability to control the page flow or navigation

- JSF works with a "managed bean" which is a regular Java class linked to JSF pages and managed by the JSF application
- JSF is the preferred way to develop Java web applications as it provides a much simpler way to implement common web application functionality

22.12 Example JSF Page

- This page links a few text boxes on the page to properties of the 'addressBean' JSF managed bean
- When the form button is clicked the 'addNewAddress' method of the managed bean is called

```
<f:view>
  <body>
  <h2>Add Address</h2>
  <h:form>
    First name:<BR>
    <h:inputText value="#{addressBean.firstName}" />
    <BR>Last name:<BR>
    <h:inputText value="#{addressBean.lastName}" />
    <BR>City:<BR>
    <h:inputText value="#{addressBean.city}" />
    <BR>
    <h:commandButton type="submit" value="Add Address"
       action="#{addressBean.addNewAddress}" />
   </h:form>
  </body>
</f:view>
```

Example JSF Page

The example code above would also require that a JSF managed bean is declared with the name 'addressBean'.

The JSF tags in the example above start with '<f:..>' or '<h:..>'. These tags must be declared in some way although there are slightly different ways to do this. When using XHTML for JSF pages the tags are part of XML syntax. When using JSP pages for JSF the tags are declared as custom JSP tag

libraries.

22.13 What is an EJB?

- Enterprise JavaBean
- Java components that provide reusable business logic or model business data
- Able to use additional services provided by the server, like distributed transactions
- Not directly accessible from a web browser
  - An EJB client will be a servlet or another Java class
  - Used as part of processing many different types of requests
- Good at: directly manipulating data, providing transactional qualities, modeling business logic

22.14 EJB Types

- Two major types of EJB
  - Session – Encapsulate business logic
    - **Stateful** – Retains information about a particular client interaction
    - **Stateless** – No information retained about client interaction
  - Message Driven – Responds to asynchronous JMS messages
    - Client interacts by sending a message
- EJB 3.0 introduced a much simplified programming model for EJB code
  - Only a few annotations are needed to turn a regular Java class and interface into an EJB

```
@Remote
public interface CustomerManager { ... }

@Stateless
public class CustomerManagerBean implements
```

```
        CustomerManager { ... }
```

EJB Types

Prior to the EJB 3 specification, there were three types of EJBs. This included an "Entity EJB" that represented persistent data. This has been replaced by the "Java Persistence API" or JPA and is discussed in the next slide.

22.15 Java Persistence API

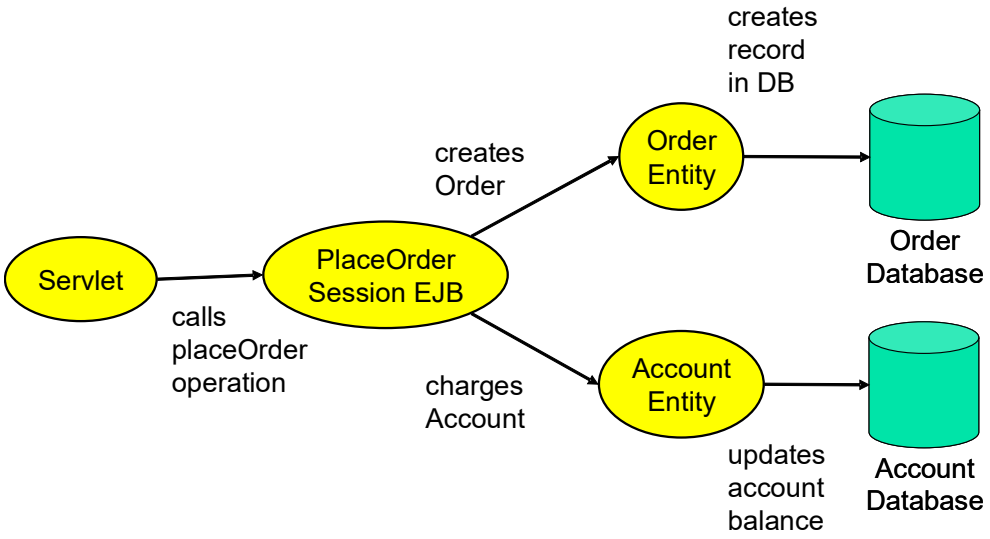
- Supported by EJB 3.0+
- Can be used as a stand-alone API with Java SE and Java web applications also
- Describes the rules to managing persistence and doing Object-Relational (O-R) mapping
- In Java Persistence API, persistent objects are referred as **Entities**
  - Only a few annotations are required to turn a Java class into a JPA entity

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements Serializable {
        @Id
        @Column(name="CUSTID")
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        int customerID;
        // other persistent properties
}
```

Java Persistence API

The Java Persistence API specification, which is part of the EJB 3 specification (JSR 220), can be downloaded from http://java.sun.com/products/ejb/docs.html.

22.16 EJB Examples



EJB Examples

The Session Bean above is used to start a transaction that includes the other components. The transaction begins with the call to the 'placeOrder' operation and ends after all the steps have been executed. This means that if there is a problem with creating the Order in the database the update to the Account database, which represents the charge, will not be saved. Instead the Account database will be restored to the state it was in before the operation began.

The above diagram also presents a common use of EJBs. Since Entities are used to modify and query data they should not be used to encapsulate complex business logic. This business logic, which includes creating an Order and charging the appropriate Account, is instead provided in a Session EJB.

22.17 Java Web Services

- Web Services can be used to execute remote operations or exchange data between different applications
- Although many technologies of web services are not specific to a particular programming language, Java has several standards to help simplify web service programming in Java
  - JAX-WS is for the "classic" SOAP/XML/WSDL web services that have been around longer
    - This was introduced in Java EE 5 to replace the more difficult JAX-

RPC web service model

- ◇ JAX-RS is for “RESTful” web services which use the HTTP specification more and have become more popular recently
  - Although introduced with Java EE 6 it is possible to use with previous versions of servers with some extra configuration

- Both are based on placing code annotations within the code to define how to expose functionality of a Java class as a web service
- You can use both styles in Java EE applications as they both have different strengths and weaknesses and are not mutually exclusive

22.18 Contexts and Dependency Injection for Java EE (CDI)

- One of the biggest added features of Java application frameworks in the past has been "dependency injection"
- In Java EE 6 a full dependency injection model was added as a standard
  - ◇ Java EE 5 had some dependency injection but was limited in what could be injected and where
- JSR 299 Contexts and Dependency Injection (CDI) allows any Java component to be injected into any other Java component
- CDI uses the injection annotations defined in JSR 330, Dependency Injection for Java
  - ◇ The JSR 330 specification was created mainly to break the dependency injection annotations out of JSR 299 so they could also be used in Java SE applications
  - ◇ Contrary to it's name though, JSR 330 is NOT a full dependency injection framework and is only the injection annotations
    - You would need a JSR 299 implementation or a non-standard dependency injection framework like Spring to interpret the JSR 330 annotations

Contexts and Dependency Injection for Java (CDI)

The addition of a standardized dependency injection model to Java EE 6 was the source of much

debate. Although probably the most popular dependency injection framework, the developers and the leaders of Spring were not even part of the initial discussions. Eventually Spring helped create the JSR 330 specification because they knew that CDI was going to be included in Java EE 6 and they did not want to have dependency injection "stolen" as a Java EE technology. JSR 330 helped retain injection annotations that could also be supported in Java SE or Spring applications.

22.19 Web Browser

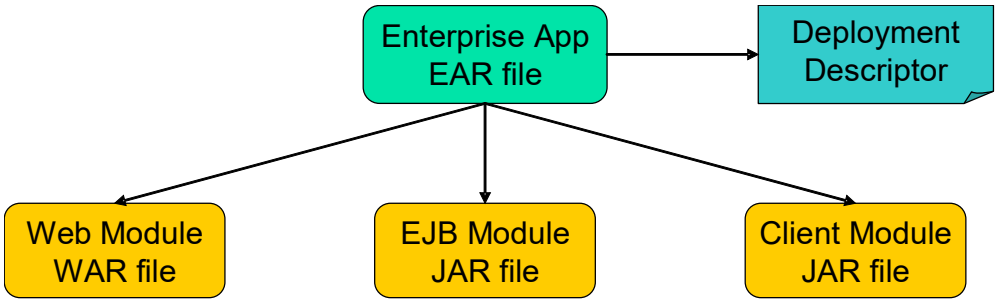
- The most common client to Java EE Enterprise Applications is a Web browser
  - “Thin Client”
- Browser sends HTTP requests to Servlets and JSPs
  - No direct interaction with EJBs
- The Servlets and JSPs interact with EJBs to access business logic and return results to the browser using HTML output
- The responses returned to the client can include JavaScript which can perform some functions like error-checking on the client but can also be disabled
- The only client requirement is support for the HTML syntax being returned

22.20 Java EE Application Structure

- The Java EE specifications also define a standard way for packaging components of an Enterprise Application
- Using this standard application structure ensures applications can be deployed to any Java EE Application Server

22.21 EAR File

- Packaging of an Enterprise Application
- Contains all components in a single archive
- Contain Modules and optional XML Deployment Descriptor
  - Always \META-INF\application.xml



EAR File

There is also a "Resource Adapter Archive" that is used to connect a Java EE application to a legacy system not directly supported by the Java EE specifications. These archives are generally provided by the vendor of that system and not developed "in house". The modules listed above would all be developed for a particular application.

22.22 What are Modules?

- Provides detailed information about individual components
  - EAR Files only contain information about Modules
  - Modules contain information about individual components
- Web Module
  - Contains resources for Servlets, JSPs, supporting Java Classes, HTML and images
  - Packaged in a .war file
- EJB Module
  - Contains EJB Java Classes
  - Packaged in a Java .jar file
- Client Module
  - Contains Application Client Java Classes

22.23 Summary

- The Java EE specifications extend the basic Java APIs

- The specifications describe an "Application Server" and components that run on that server
- The Java EE specifications provide a robust and standard environment for enterprise-scale Java applications

Chapter 23 - Appendix: Advanced Java Tools

Objectives

- Cover some of the advanced Java refactoring tools provided by Eclipse
  - Refactoring defined
  - Renaming elements
  - Moving a class to a different package
  - Extracting code to a method
  - Refactoring the type hierarchy
  - Generalizing a variable
  - Pull-up and push-down methods

23.1 Refactoring

- Wikipedia definition - **Refactoring is the process of rewriting written material to improve its readability or structure, with the explicit purpose of keeping its meaning or behavior**
  - For example, rename a method or class
- Eclipse makes workspace-wide changes so that any compilation problem is avoided
  - This greatly simplifies refactoring
- Use refactoring to clean up code, adhere to better naming conventions, and reorganize classes in different projects or packages
- Refactoring commands are available from the context menus of several Java views (e.g. Package Explorer, Outline) and editors

23.2 Renaming Elements

- Many elements can be renamed with the tools of Eclipse
  - Classes

- Methods
- Variables

- This is done so the new name is more descriptive of the purpose
  - Renaming can also update all references so the code still compiles
- To do this you can select or right-click the element in one of the Java views or in the source editor and select **Refactor → Rename**
  - Renaming a local variable or method parameter must be done in the source editor

23.3 Moving a Class to a Different Package

- You can use refactoring to move a resource between Java packages
  - In the Package Explorer view or Project Explorer view, select the <classname>.java file from the current package and drag it into the <new package> package
  - You can also select the file and choose **Refactor → Move** from the context menu.
    - You are prompted to select whether or not to update references to the file you are moving. Typically, click **OK**, to avoid compile errors
    - Use the **Preview** button to see the list of changes that will be made as a result of the move
    - Click **OK**
- The file will be moved, and its package declaration changes to reflect the new location

23.4 Extracting Code to a Method

- Extracting a range of code to a new method has the following benefits
  - It breaks a large method into smaller logical methods
  - It facilitates reuse. The new method can be called anywhere within the program to execute that piece of code

- Eclipse can infer parameters and return type of the new method based on the code highlighted
- Not all code can be extracted into a separate method
- To extract a block of code into a separate method
  - In the editor, select the block of code that needs to be extracted being careful to select a complete block of statements
  - From the selection's pop-up menu, choose **Refactor → Extract Method....**
  - In the Extract Method dialog, enter the new method name, change the access modifier, and modify the parameter definitions
  - You can preview the changes with the **Preview** button and click **OK** to create the new method
  - View the extracted method by selecting it in the Outline view

Extracting Code to a Method

If two or more variables change with the code selected and are referenced after the selected code the method refactoring won't work as a method can only return one value

23.5 Other Source Code Refactoring

- **Extract Local Variable**
  - Select an expression that can be resolved to a single value and select the **Refactor → Extract Local Variable** and provide a name for the new variable
  - The code will add a declaration of that variable initialized to the value of the selected expression
  - Occurrences of the expression can be replaced by the new variable
- **Extract Constant**
  - Similar to extracting a local variable but declares a constant
- **Inline**
  - Inline is the reverse of 'Extract Local Variable' and removes a variable

and replaces use of the variable with the expression the variable was initialized with

- **Convert Local Variable to Field**
  - Turns a local variable declared in a method into a field in the same class
  - If the local variable was initialized with an expression that expression is used to initialize the field

23.6 Refactoring to Improve Type Hierarchy

- Often as applications are developed opportunities to leverage inheritance and interfaces is discovered
  - Eclipse refactoring tools can help create the appropriate hierarchy and update existing code
- **Extract Class** can take a selected group of fields and create a new class which encapsulates those fields
- **Extract Superclass** can create a new superclass from existing classes and modify those classes to be subclasses of the new superclass
  - Existing code can be examined to see if the new superclass can be used instead of one of the more specific subclasses
- **Extract Interface** can define a new interface with selected methods from a class and modify the class to implement the interface
  - Existing code can be examined to see if the new interface can be used where the methods added to the interface are used

23.7 Generalizing a Variable

- Often generalization in a design is recognized at a late phase
  - For example, common elements of the SavingsAccount and CheckingAccount classes are moved to the Account class
  - Code that had been using objects of the SavingsAccount type can now be written using the more general Account type

- To reflect this design change, you may need to use a more generalized data type in the code
  - For example, change the data type of a variable from SavingsAccount to Account
- Right click on a member variable in the Outline view and select **Refactor → Generalize Declared Type**
  - Choose from a list of all known super classes and implemented interfaces for the current data type

23.8 Pull-up and Push-down

- **Pull Up** moves member variables or methods to the class's superclass:
  - In a Java view (e.g. Outline, Package Explorer, Members view), select the members that you want to pull up
  - Do one of the following:
    - From the menu bar, select **Refactor → Pull Up**
    - From the pop-up menu, select **Refactor → Pull Up**
- **Push Down** moves member variables or methods to all known subclasses of a class
  - Similar steps as above
- As an example say that the SavingsAccount and CheckingAccount subclasses of the Account class both have an 'id' property
  - The 'Pull Up' refactoring could move the definition of this property into the Account class and remove the definition from the subclasses

Pull-up and Push-down Methods

Note: the selected members must all have the same declaring type for this refactoring to be enabled.

The Pull Up and Push Down refactorings are useful if you want to change the structure of superclass and subclass after they both exist.