Presentation Slides:
For INSTRUCTOR Use Only

Fast Track to Java 8 and OO Development

Version: 20160729

Course Overview

- An introductory Java course that starts with basic principles
  - Provides a solid foundation in the concepts and practices for writing good object-oriented systems in Java
  - Provides knowledge needed to productively use core Java technology for programming
  - Including database access with JDBC/JPA
- Be prepared to work hard and learn a great deal!
- The course contains numerous hands-on labs
  - They exercise all the important concepts discussed
  - The lab solutions for the course are provided to you
- The course covers all core features of Java
  - It supports all recent versions of Java

Course Objectives

- Learn Java's architecture and uses
- Understand Java language basics
- Compile and execute Java programs with development tools such as `javac` and `java`
- Learn object-oriented (OO) programming and the object model
  - Understand the differences between traditional programming and object-oriented programming
  - Understand important OO principles such as composition inheritance and polymorphism
- Use Java packages to organize code
- Understand interfaces, their importance, and their uses

Course Objectives

- Learn (and practice!) Java naming conventions and good Java coding style
- Create well-structured Java programs
- Use core Java API class libraries
- Understand how exceptions are used for error handling
- Understand the basics of database access with JDBC and JPA
- Learn the basics of the Collections Framework
- See some of the new/advanced Java language features, including Java 8 lambda expressions
- Understand and use basic I/O streams (optional)

Labs

Lab

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - The detailed lab instructions **are separate** from the main lecture manual
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

  `JavaInstructor teacher = new JavaInstructor()`

  –Code fragments use the same font, e.g. `teacher.teach()`

  –We **bold**/color text for emphasis

  –Filenames are in italics, e.g. *JavaInstructor.java*

  –Notes are indicated with a superscript number [1] or a **star ***

  –Longer code examples appear in a separate code box (below)

  - Code fragments may leave out detail (e.g. imports, or class def)

```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
   public void teach() {
      System.out.println("Java is way cool");
   }
}
```

Course Outline

- Session 1: **A Simple Java Program and the JDK**
- Session 2: **Java Overview**
- Session 3: **Class and Object Basics**
- Session 4: **More on Classes and Objects**
- Session 5: **Flow of Control**
- Session 6: **Strings and Arrays**
- Session 7: **Packages and Access Protection**
- Session 8: **Composition and Inheritance**
- Session 9: **Interfaces**
- Session 10: **Exceptions**
- Session 11: **Collections and Generics**
- Session 12: **Database Access with JDBC and JPA**
- Session 13: **Additional Language Features**
- Session 14: **Java I/O** (optional)
- Appendix: **JDBC**

Session 1: A Simple Java Class and Running a Java Program

Session Objectives

- Look at a simple Java program, and how it is compiled and run
- Set your computer up for Java development
  - Setting paths, environment variables, etc.
- Use a text editor to type in a simple Java class
- Use the Java Development Kit to compile and run the code

A Simple Application Class

- We'll first review a very simple Java app
  – It displays "Hello World"
  – It's a non-graphic standalone application
- All Java programs are a **`class`** with a **`main`** method in it
  – The HelloWorld source file must be named ***HelloWorld.java***
- **`main()`** is a special method that is the starting point of an app
  – It must be declared as below, and has to appear in a class
  – `System.out.println` outputs to the console

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

HelloWorld in More Detail

- We'll cover these details in more depth later

Comment

Class definition

Method definition

Semicolon

Block

```
// a simple application to display "Hello World"
public class HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println("Hello World");
        }
}
```

Method call

Method argument

Compiling `HelloWorld`

- To use a Java class, **compile** the Java code with **`javac`**
  - This is the Java compiler

```
C:\StudentWork>javac HelloWorld.java
```

  - Compiling creates ***HelloWorld.class***
    - Which contains Java **bytecode**

- Use the Java Virtual Machine (**JVM**), to run the program
  - Via the **`java`** executable

```
C:\StudentWork>java HelloWorld
```

  - Producing "Hello World" in your console

Note on Comments

- Java has 3 kinds of comments
  - Single line comment: Starts with **// ...**
  - Multi-line using **/* ... */**
  - Javadoc comments using **/** ... */** (1)

```
/**
 * This class prints "Hello World" to standard output
 */
class HelloWorld  {   // this comment starts mid-line
  public static void main(String[] args) {
    System.out.println("Hello World");
    // Easy to comment out code this way
    // System.out.println("Bye");
  }
}
/* This class still needs work.  We need to add many
   interesting things, and show how cool Java is.  */
```

Lab 1.1: HelloWorld

In this lab, we will compile and run a very simple Java program

Review Questions

1. What is the purpose of the `main` method?

2. How do you print something to standard output (the console)?

3. What tool do you use to compile Java source code?

4. What tool do you use to run compiled Java code?

Lesson Summary

- The `main()` method is the entry point for all Java programs
  – It appears in a Java class definition

- `System.out.println()` is one simple way to print to the console
  – It is part of the standard Java libraries

- Java source code is created in a file with a **.java** extension
  – It is compiled with the `javac` program

- Compiled programs are run with the `java` program
  - The Java virtual machine, or JVM

Session 2: Java Overview

Language and Platform Features

Program Life Cycle

The Java SE Development Kit (JDK)

Session Objectives

- Discuss Java's advantages
- Understand how Java is a language and a platform
  - Know how it supports multiple environments
  - Be aware of the different Java platforms
- Define portability and explain how Java achieves this
- Know the Java program development and runtime lifecycle
- Understand the JDK and be ware of some of its tools

Language and Platform Features

Language and Platform Features

Program Life Cycle

The Java SE Development Kit (JDK)

What is Java?

- **A programming language**
  - A strongly typed, object-oriented, general purpose programming language
- **A runtime system**
  - The JVM translates Java bytecode (output from the compiler) to native operating system code at runtime
- **A platform**
  - JRE, the Java Runtime Environment, contains the JVM + other runtime facilities
    - A JVM is available for almost every popular operating system
  - The Java API, including many standard classes
    - e.g., `String` and `System`

Java is Modern and Object-Oriented

- Supports many modern features
  - Networking, multithreading, database access, exceptions, internationalization, security, data structures, GUI, and more
- Java is **Object-Oriented**
  - **Object-Oriented Programming** is a way of thinking about and modeling systems
- **OO programming** creates types which model the real world
  - Forming an abstract blueprint of your system
    - e.g., Customer, Employee, PurchaseOrder, ShoppingCart, etc.
  - Types have properties (data) and methods (behavior)

Java is Portable and Safe

- Java programs are **platform independent**
  - Java source compiles to the same **bytecode** across all platforms
  - Bytecode executes in the JVM at runtime
    - Where it's translated to Windows code, UNIX code, etc.
  - Java programs run anywhere there is a JVM
- Java eliminates **error prone** features
  - e.g., pointer arithmetic and "go to" statements
- Java **helps you develop reliable software**
  - Garbage collection to prevent memory leaks
  - Array bounds checking
  - Compiler-checked exceptions
  - Removal of unsafe capabilities (e.g. raw pointer access)

Java has Multiple Platforms

- **Java Standard Edition (Java SE)**
  - Core APIs and some enterprise APIs (e.g. Web Services)
  - **Java SE Embedded** and Java 8 "**compact**" profiles define subsets of Java SE with a reduced footprint

- **Java Enterprise Edition (Java EE)**
  - Platform for multitier enterprise applications
  - Depends upon Java SE
  - Adds enterprise capabilities like Web apps, messaging, etc.

- **Java Micro Edition (Java ME)**
  - Platform for small devices like cell phones
  - Subset of Java SE, with a smaller footprint
  - Lower resource usage than Java SE Embedded

- Android, a de-facto standard, is a different platform altogether (1)

Program Life Cycle

Language and Platform Features

Program Life Cycle

The Java SE Development Kit (JDK)

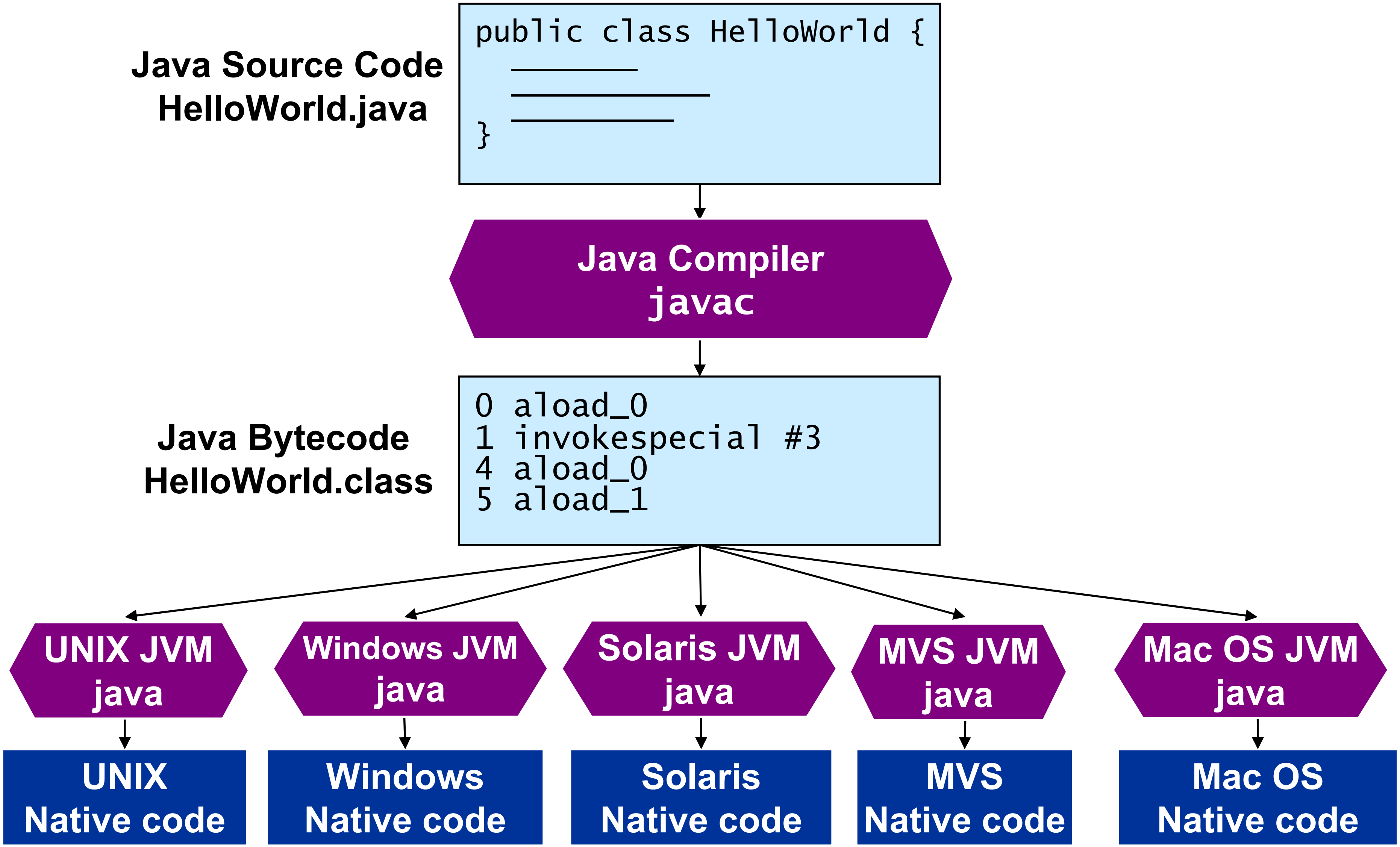
Java Source and Java Bytecode

- Java programs are stored in a .java file, e.g. ***MyClass.java***

- They are compiled into an intermediate language
  - **Java bytecode**, stored in a **.class** file, e.g. ***MyClass.class***
  - Bytecode is platform independent and interpreted by the **JVM**
  - Think of bytecode as the native instructions for the JVM
  - Bytecode helps make "**write once run anywhere**" possible

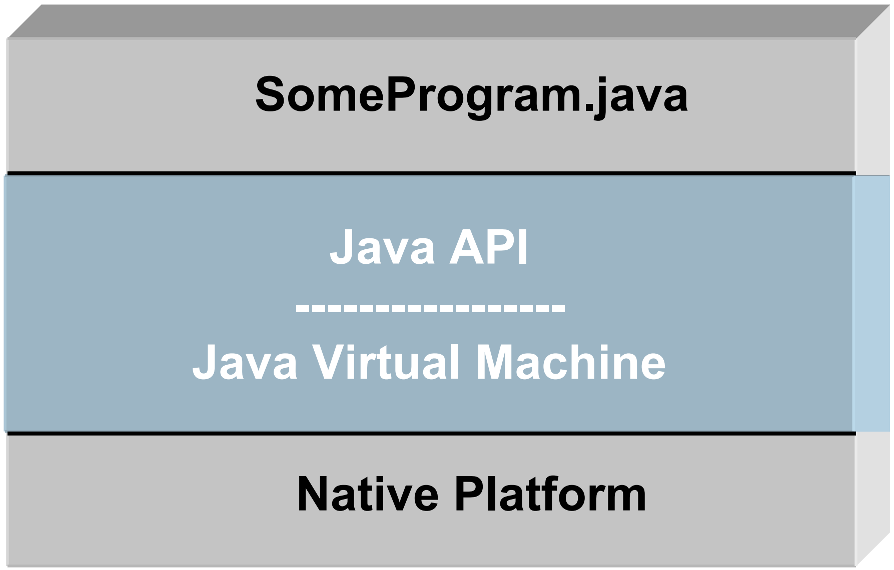
- The JVM converts bytecode into the **native code** for the target environment
  - It then runs the program

Life Cycle of a Java Program



Java Programs Insulated From Environment

- Java provides a platform to run programs independently of the environment
  - A software-only platform of the Java API and JVM



Java is Dynamic - The Runtime Process

- Several phases take place at runtime

Java Bytecode
HelloWorld.class

Includes symbolic reference information from the compiler

Java Class Loader
`java`

Loads bytecode into memory

Java Verifier
`java`

Confirms that bytecodes are valid and follow security rules

Java Interpreter
`java`

Translates bytecode into native code and performs final name resolution (linking)

The Java SE Development Kit (JDK)

Language and Platform Features

Program Life Cycle

The Java SE Development Kit (JDK)

Java Development Kit (JDK)

- The **J**ava SE **D**evelopment **K**it (or "JDK") provides all basic components for Java programming

- Major components include:
  - The Java core API libraries
  - Command line tools

`javac`	**compiler**
`java`	**runtime system**
`javadoc`	**documentation tool**
`jar`	**creates JAR (Java ARchive) files**
`javap`	**allows you to peek inside .class files**

The Java API

- Java provides a large **Application Programming Interface** (**API**)

- Grouped into **packages** (related libraries), including:
  - **Core** (java.lang): Common functionality such as the `String`, `Double`, and `Exception` classes
  - **Utility** (java.util): Collections, internationalization, date/time, and logging
  - **I/O** (java.io): File I/O, buffers, File objects, abstract channels
  - **Persistence** (java.sql, javax.persistence): Database Access
  - **Networking** (java.net): Including TCP/IP and SSL
  - **GUI** (java.awt, javax.swing): Graphical User Interfaces (GUI)
  - **XML**(javax.xml, javax.jws): XML manipulation and Web services

Downloading and Installing the JDK

- Try the following:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

- For Java 8 (Based on JDK 8 update 91 - 64 bit install)
  - Install file: `jdk-8u91-windows-x64.exe`
  - Default installation directory:
    *C:\Program Files\Java**jdk1.8.0_91***

- **NOTE:** the core API documentation is not included
  - It can be downloaded separately as a zip file
  - Unzip it into the Java installation directory

Lab 2.1: The Development Environment

In this lab, we'll become familiar with the lab development environment

Review Questions

1. Explain the Java program development and runtime lifecycle.
2. What is portability? How does Java achieve it?
3. What is the role of the Java virtual machine?
4. What is the JDK? Name some of its tools and what they are used for.

Lesson Summary

- Java source files have a ***.java*** extension
  - They're **compiled into bytecode** (*.class* files) with javac
  - They **run in the JVM**, which converts them to the target platform
  - The JVM provides an **independent operating environment** for Java programs

- Java programs can run unchanged in different environments and operating systems.
  - Done by using the **same bytecode format** and **standard libraries** across all platforms

- The JDK provides tools for working with Java
  - Including **`javac`** (the java compiler) **`java`** (The JVM), and others

Session 3: Class and Object Basics

Object-Oriented Programming Overview

Classes, References, and Instantiation

Methods and Data in a Class

Session Objectives

- Provide an **Object-Oriented Programming** (OOP) overview
  - List some of the advantages and principles of OOP
- Discuss the major **characteristics of an object**
- Explain the difference between a **class** and an **object**
  - And the difference between an object and its references
- Create/test a Java **class definition**
  - Including instance variables, getter/setter methods, and constructors
- Explain what **encapsulation** is and why we use it
  - Explain and implement encapsulation in Java

Object-Oriented Programming Overview

Object-Oriented Programming Overview

Classes, References, and Instantiation

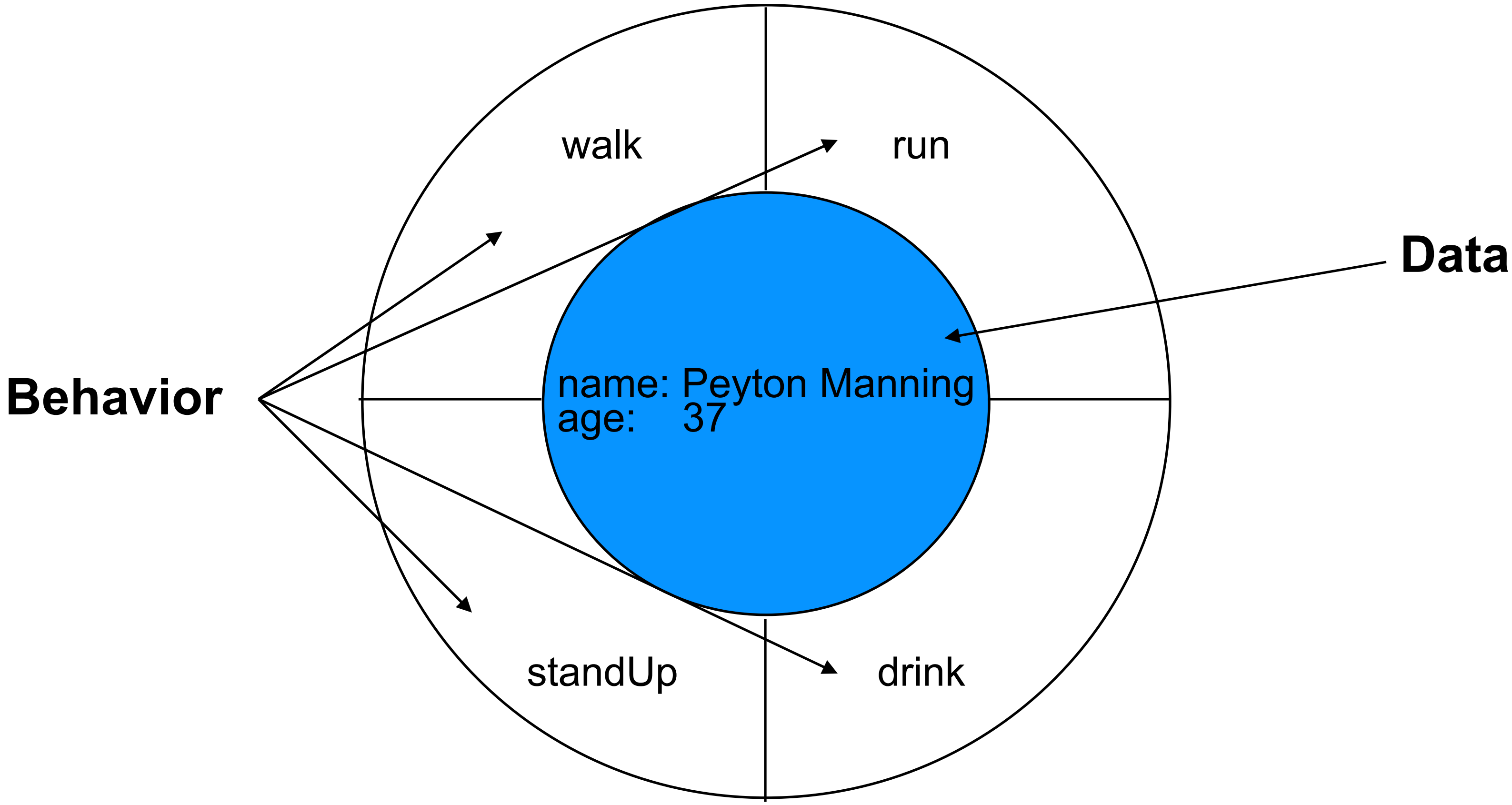
Methods and Data in a Class

What is Object-Oriented Programming?

- A way of thinking about and modeling systems
  - One of many methodologies within software engineering
  - A scientifically sound approach with a proven track record
- Object Orientation
  - **Ties data together with the code** that manipulates it
  - Organized in modules called **classes** or **types**
- A type can **represent** a **concept**, **abstraction**, or **thing**
- Objects (instances of a type) **work together** to perform the functionality of software systems

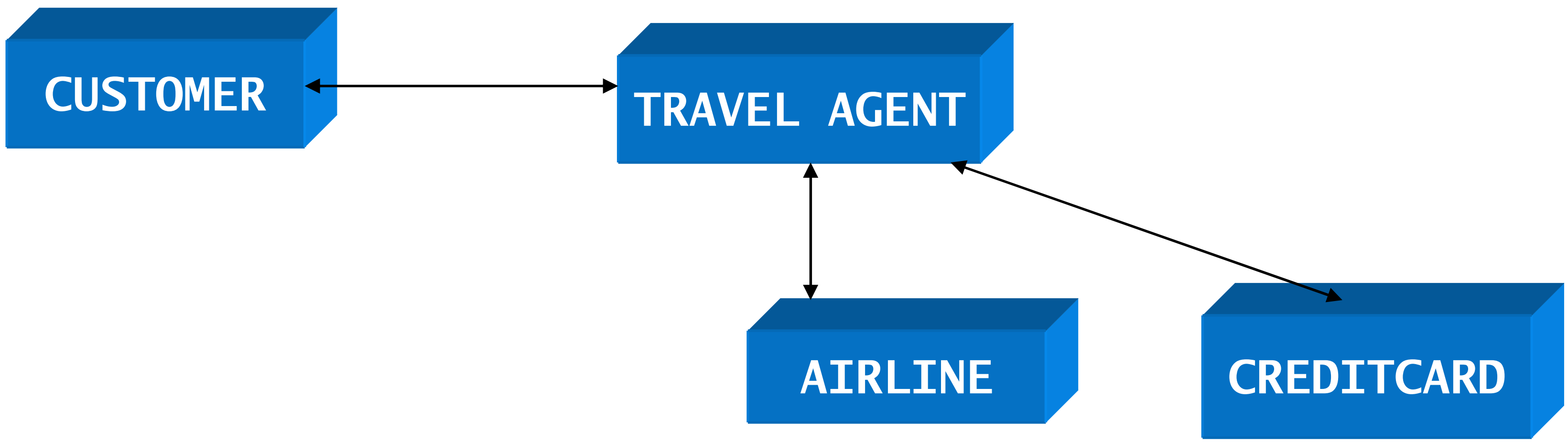
What is an Object?

- A software representation containing **data** and **behavior**
  - – e.g. a person's name and age, with various behavior



Important Object Characteristics

- **Type or Class**: The kind of thing an object is - its classification
- **Data or Properties**: Information associated with an object
- **Behavior or Methods**: Actions that apply to an object
- **Identity**: The unique existence of every object, independent of its characteristics
- OOP, organizes a program around **objects**, rather than functions or processes
  - An object-based system is a set of collaborating objects:



About Object-Oriented Programming (OOP)

- **Six Core Principles** of OOP
  1. Everything is an object
  2. Objects perform work by making requests of each other
  3. Every object has its own data, which may consist of other objects
  4. Every object is an instance of a type – a type groups similar objects
  5. The type is the repository for an object's behavior
  6. Types are organized into an inheritance hierarchy

- Some advantages of OOP
  - Handles large, complex systems better
  - More flexible, maintainable, and reusable code
  - Models the real world - better accuracy in modeling
  - Scalable - ability to grow with the organization
  - Faster implementation (large API, simpler syntax, portability)

What's a Type?

- Generally, the way that you classify objects, for example:
  - **Concrete**: Person, Car, Planet, Star
  - **Concepts**: Number, Democracy
  - **Events**: General Protection Fault, Earthquake
- The types you use depend on your problem domain
  - In Java, types are represented by classes and interfaces
- Determining your types is a key step in creating a system
  - What the types are is not a clear-cut, black-and-white decision
  - Discovering them, along with their interactions, is a creative process

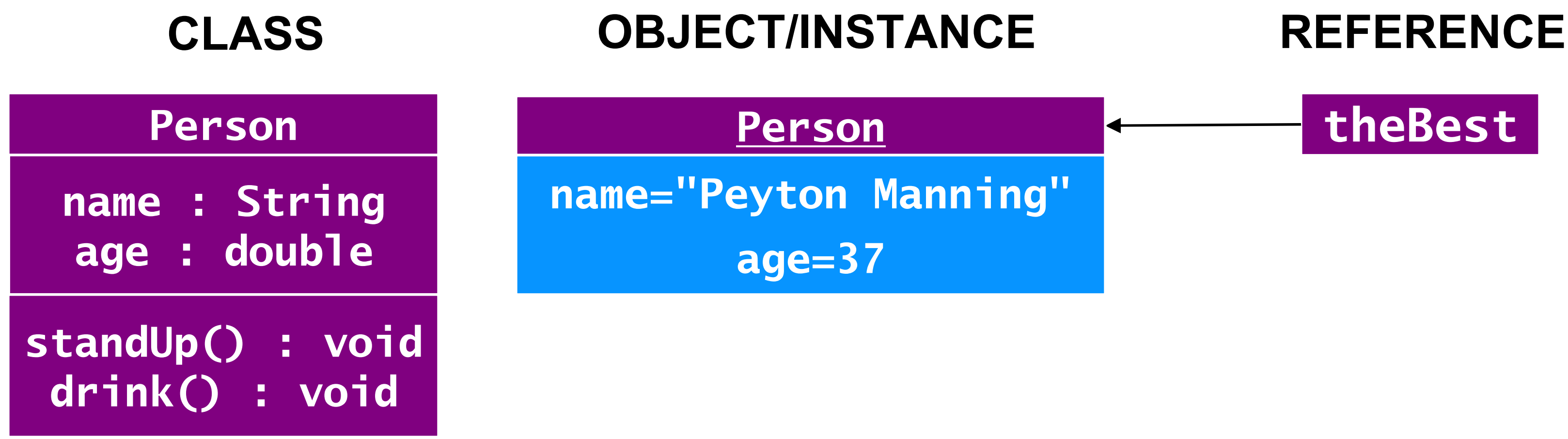
Types, Instances, and Property Values

- A type is a **blueprint** for objects (but not the object)
  - A person's definition (the `Person` type) is not actually a person
- Actual objects are called **instances** of their type(s)
  - You, the students, are instances of the `Person` type (one would hope)
  - Creating instances of a type is called ***instantiating*** that type
- Each instance has its own values for its properties
  - You all have your own names, ages, incomes, etc.

Classes and Objects

- A **class** defines a new type in Java
  - It's the **blueprint** describing its properties and behavior
  - Properties hold **information**
  - Methods perform the **behaviors** of an object

- An object, or instance, is the actualization of a class
  - As shown in the UML-based diagrams below [1]



Lab 3.1: Exploring Types and Objects

In this (discussion only) lab, we'll explore the notion of defining types

Identity and Object References

- Each object has a unique existence or ***identity***
  - – Independent of any properties
  - – A property may be a unique identifier, but that's separate from the identity
    - e.g., a social security number
  - – A person's existence is not dependent on their SS number

- ***Object references*** refer to objects
  - – They are handles on objects
  - – We use them to access and interact with objects

Lab 3.2: Identity and Object References

In this (discussion only) lab, we'll explore the notion of object instances and references

LEARNINGPATTERNS

Classes, References, and Instantiation

Object-Oriented Programming Overview

Classes, References, and Instantiation

Methods and Data in a Class

The Class in Java

- A **class** defines a new type
  - Defined in a ***class definition***
  - It may contain both data and methods

- It's a **central concept** in Java
  - Data is defined in its **fields**, or **instance variables**
  - Behavior/code is defined in its **methods**

- There is no global data, nor global methods

Class Definition

- This is a simple class definition

```
// define a class to represent an alarm clock
public class AlarmClock
{

}
```

- The `class` keyword introduces the class
  - `AlarmClock` is the name of the class
  - `{` starts the class body
    - Data and methods are declared within the body
  - `}` ends the class definition
  - We'll talk about the `public` keyword later

A Class Definition is a Blueprint

- A class definition is a ***blueprint*** to create instances
  - Writing it doesn't create instances
  - It just shows what they'll be like when you do create them

- A class definition is like a **cookie cutter**
  - **It's not a cookie**
  - To make a cookie, you cut cookie dough with the cutter
  - To make an object, we cut one out from object dough (memory)
    - With the class definition as the cookie cutter

- Creating the object is also known as ***instantiating*** the class

Creating and Referencing Objects

- Use **new** and the class name to ***instantiate*** instances
  - `new AlarmClock()` creates an instance of `AlarmClock`
  - It is created on the "heap" (1)
- Objects live in a computer's memory
  - Unlike cookies on a cookie sheet that we can pick up and eat
  - How do we pick up the `AlarmClock` instance? With variables.
- **Variables** can reference an object instance
  - They have a **name** and a **type**
  - Below, the left hand side declares an `AlarmClock` variable
  - It refers to a newly created `AlarmClock` instance

```
AlarmClock anAlarmClock = new AlarmClock();
```

More About Identifiers

- ***Identifiers*** start with a **letter**, **underscore _**, or **dollar sign $**
  - Subsequent characters can be digits
  - Useable as variable names, class names, method names, etc.

- **Java naming conventions** (recommended, not required)
  - **Classes**: Generally **nouns**, spelled as **CamelCase**
    - `String`, `System`, `HelloWorld`, `TextField`
  - **Variables**: Spelled as **camelCase**
    - `salary`, `hairColor`, `maxUpdateInterval`
  - **Method**: Generally **verbs**, spelled as **camelCase**
    - `getSalary()`, `dye()`, `setHairColor()`, `service()`

Methods and Data in a Class

Object-Oriented Programming Overview

Classes, References, and Instantiation

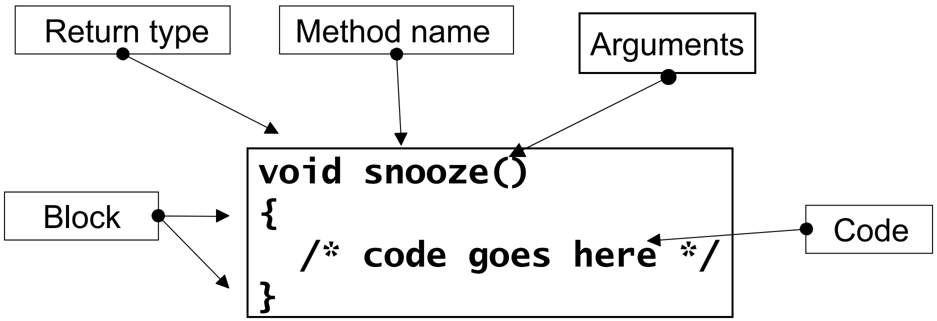
Methods and Data in a Class

Behavior and Methods

- Behavior is defined in ***methods*** within the class body
  - They associate executable code with classes

```
public class AlarmClock {
  void snooze() {
    System.out.println("ZZZZZ");
  }
}
```

- Methods can return values
  - A `void` return means nothing is returned
- Methods can be passed arguments (covered later)



Invoking Methods

- Invoke methods with an instance, followed by the ***dot operator .*** and the method name
  - You **always** need an instance for a regular method

`referenceVariableName.methodName()`

```
// Here's a program that uses AlarmClock

public class EarlyMorning
{
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    AlarmClock yourClock = new AlarmClock();
    myClock.snooze();
  }
}
```

Storing Data in Objects

- A Class declares data in its ***fields*** (or **instance variables**) (1)
  - Each instance stores its own values
  - These values can **vary** from **instance to instance**

- If not initialized, fields are initialized to a **default value** (2)
  - Zero for numeric values, null for references

```
// class AlarmClock with instance variable for snooze time
public class AlarmClock {

  // Variable declaration
  int snoozeInterval;
}
```

About Fields

- Each instance contains a copy of its fields
  - When an instance is created, space is allocated for all its fields
  - This makes sense – every person has their own name, each clocks stores its own time

- You can initialize a field as shown below
  - Instead of using the default value

```
public class AlarmClock
{
  // declare and initialize a field/instance variable
  int snoozeInterval = 5;
}
```

Data Access and Return Values in Methods

- Methods can access instance variables
  - Use the name of the variable (`snoozeInterval`)
  - The data used is from the instance that invokes the method
    - Since there is **always** an instance used with a normal method call

- Methods return values using the **`return`** keyword
  - The returned value must be consistent with the method's return type

Data Access and Return Values in Methods

```
public class AlarmClock {
  int snoozeInterval = 5;

  int getSnoozeInterval() {   // Must return an int
    return snoozeInterval;   // Return the instance var's value
  }

  void snooze() { /* … */ }
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    // …
    int snoozeInterval = myClock.getSnoozeInterval();
    System.out.println("More sleep:" + snoozeInterval);
    myClock.snooze();
  }
}
```

Accessing Data (Another Way)

- Outside the class, you can access a field through an instance
  - Via the ***dot operator*** **.** with the instance
  - We need an instance - it doesn't make any sense to say:
    - What is `AlarmClock`'s snooze interval?
  - We need a specific `AlarmClock` object to get its data

- The example below illustrates this
  - But is **NOT** good practice - we'll discuss this later

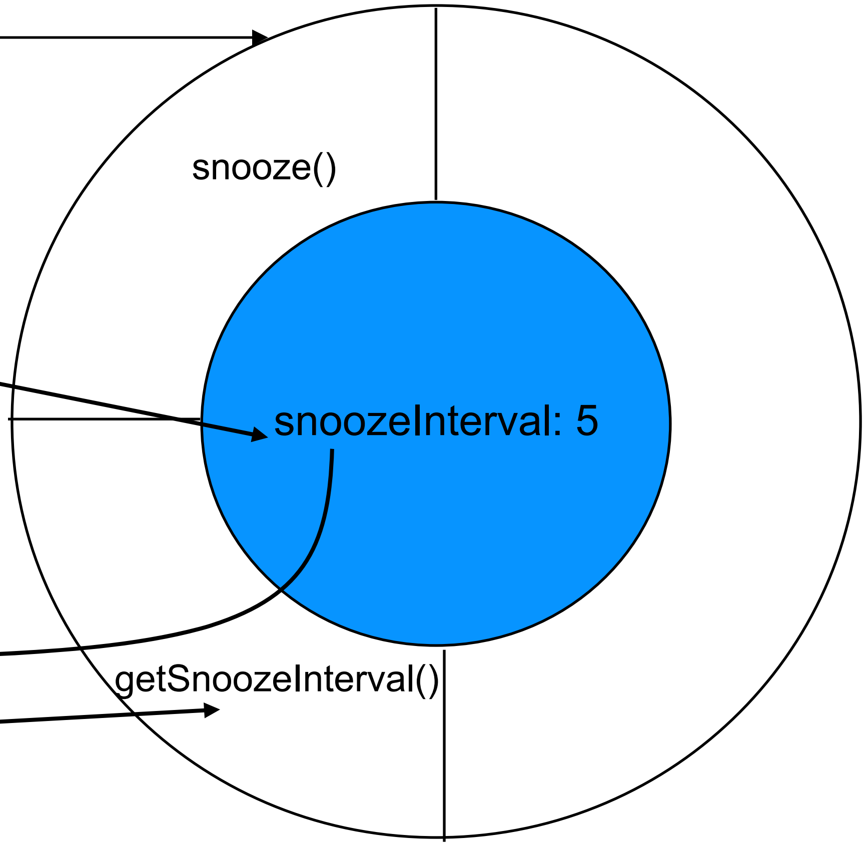
```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    // …
    int snoozeInterval = myClock.snoozeInterval;
  }
}
```

Pretty Pictures

```
AlarmClock myClock = new AlarmClock();
```

```
int snoozeInterval =
      myClock.snoozeInterval;
```

```
int snoozeInterval =
  myClock.getSnoozeInterval();
```



More About Variables

- Variable declarations have the basic form:

  TypeSpecifier Name = VariableInitializer$_{opt}$

```
int n;                    // no initialization
int j = 0;                // with initialization
int k = j + 1;            // initialize with expression
float r = 0.0F;           // float (decimals default to double)
```

  - The type indicates the kind of data and legal values
    - Sometimes called the *compile-time* type
  - Variable assignments are checked at compile time
    - To make sure a compatible value is assigned
- The lines of code above are ***statements***
  - The smallest independent units in a Java program
  - Simple statements like these **end with a semicolon**

About Java Primitive Data Types

- All data in Java has a type
- The possible types are:
  - Class or interface type, e.g., `String`
  - A **primitive type**:

Type	Size (bits)	Range
`byte`	8	-128 to 127
`short`	16	-32,768 to 32,767
`int`	32	-2,147,483,648 to 2,147,483,647
`long`	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	32	±1.4E-45 to ±3.40E+38
`double`	64	±4.9E-324 to ±1.797E+308
`char`	16	Unicode character set
`boolean`		`true` or `false`

Numeric Literals

- **Integer** literals can be:
  - **Decimal**/Base10 (default) — `16`
  - **Octal**/Base8, with a prefix of **`0`** — `020`
  - **Hexadecimal**/Base16, with a prefix of **`0x`** or **`0X`** — `0x10`
  - **Binary**/Base 2 with prefix of **`0b`** or **`0B`** (Java 7+) — `0b10000`
- **Floating point** literals can be:
  - Decimal — `31.38`
  - Scientific notation, using **`E`** (preferred) or **`e`** — `3.138E+1`
- Default for integer literals: **32-bit `int`**
- Default for floating point literals: **64-bit `double`**
  - Use **F**, **D** to explicitly indicate `float`, `double` values (`2.5`**F**, 1.2**D**)
  - Use **L** to explicitly indicate `long` values (10**L**)

Non-Numeric Literals

- **`boolean`** literals are `true` or `false`

- **`char`** literals can use as a character in single quotes: **`'?'`**
- Or 4-digit hexadecimal Unicode character number: **`'\u0063'`**

- Escape sequences can be used for some special characters
  - **`\n`** for a new line and **`\"`** for a literal double-quote
  - Others are listed in the notes below

- Java allows underscores between numeric literal digits
  - For readability - there are restrictions (1)

```
long creditCardNumber = 1234_5678_9012_3456L;
```

Strings

- String literals are sequences of doubly-quoted characters:

`"Hello World"`

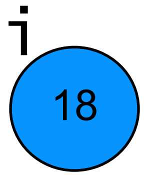
- Java handles strings in a special way
  - –Class `String` defines the string representation
  - –Declare a string reference with `String`
    - Can initialize it with a **string literal**
    - Doesn't need the **new** keyword

```
String hi = "Hello";
```

Primitive Types are Value Types

- Primitive types are **value** types
  - While object references refer to something holding the data
  - Variables of a primitive type, **hold the value itself**
  - The variable name is an alias for the chunk of memory holding the data
  - For example, the below: looks like this in memory:

```
int i = 18;
```



Arithmetic Operations

- Java provides standard arithmetic operations, including:
  - Addition (+), Subtraction (-), Multiplication (*), Division (/)
  - String concatenation (+)
  - Numerical Equality (==)
  - Bitwise operators
- Java also has logical operators, that include:
  - Logical OR (| and ||) Logical AND (& and &&)
  - NOT (!)

Primitive Type Conversion and Casting

- Conversion / casting take place at compile time
  - **Upcasts** are implicit and automatic (from smaller to larger size)
  - **Downcasts** must be made explicitly (from larger to smaller size)
    - Because you might lose data and/or precision

```
float diameter = 6.77F;
float pi = 3.1415927F;
double circum = pi * diameter;  // auto "upcast" to double

float approx = (float) circum;  // explicit "downcast"
```

- Automatic "upcasts" may occur during operations
  - Integers converted to floating points
  - Smaller size types converted to larger size types

```
int   + float  -> float        // int converted to float
float + double -> double       // float converted to double
```

Lab 3.3: Writing a Class Definition

In this lab, we will create a class that has methods and instance variables

Review Questions

1. What's a Java class, and how is it different from an object instance?
2. What's the difference between an object instance and its references?
3. What are some characteristics of an object instance?
4. What are the default values for fields (instance variables) that are not explicitly initialized?

Lesson Summary

- A **class** (or type) is a **blueprint** for objects – it is not the object
  - An **instance** is an **actual object** of some type
- An object instance (the actual object) can be **referred to** with a **reference**
  - This is a **handle** on the object
- Every object instance has a type
  - The type defines the **data** it can hold and its **behavior**
  - Each instance has its **own data**, and a **unique identity**
- Fields (instance variables) have **default values** if not initialized
  - **Zero** for numerics, **false** for booleans, and **null** for references

Session 4: More on Classes and Objects

Working with Methods and Data
Encapsulation and Access Control
Constructors
`static` or Class Members
Type-Safe Enums
Other Language Details

Session Objectives

- Understand and use encapsulation
- Understand and use constructors
- Create and test a class definition in Java
  - Including instance variables, getter/setter methods, and constructors
- Explain what encapsulation is why we use it, and how we implement encapsulation in Java
- Explain the concepts behind class (`static`) variables and class (`static`) methods, and be able to use them in your code

Working With Methods and Data

Working with Methods and Data

Encapsulation and Access Control

Constructors

`static` or Class Members

Type-Safe Enums

Other Language Details

Working Within Methods

- Methods define behavior in a class
  - They often manipulate data
- Methods generally work with:
  - Fields/Instance Variables, Method Parameters, and Local Variables
- **Method parameters**: Pass data to a method
  - Specify method parameters with a **type** and a **name**
  - Methods can access the parameters and fields

```
public class AlarmClock {
  int snoozeInterval = 0;

  void setSnoozeInterval(int snoozeIn) {
    snoozeInterval = snoozeIn;
  }
}
```

Calling Methods

- Pass data via a method's parameters when you call the method

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();

    // Pass in a value for the parameter when invoking
    myClock.setSnoozeInterval(1000);
  }
}
```

- Multiple parameters use a comma-separated list
  - e.g. for using two values for snooze interval, as below

```
// In the AlarmClock class
void setSnoozeInterval(int hour, int minute) { … }
```

```
// Elsewhere in code where you have myClock ref
myClock.setSnoozeInterval(1, 30);
```

Local Variables

- ***Local*** variables are declared in a method
  - Accessible only within the method
  - Also called ***automatic*** variables (1)

```
public class AlarmClock {
  int snoozeInterval = 0;

  // set method now returns previous value
  int setSnoozeInterval(int snoozeIn) {

    // oldSnooze is a local variable
    int oldSnooze = snoozeInterval;

    snoozeInterval = snoozeIn;
    return oldSnooze;
  }

  int getSnoozeInterval() {
    return snoozeInterval;   // Returns the field
  }
}
```

The this Variable and Instance Data

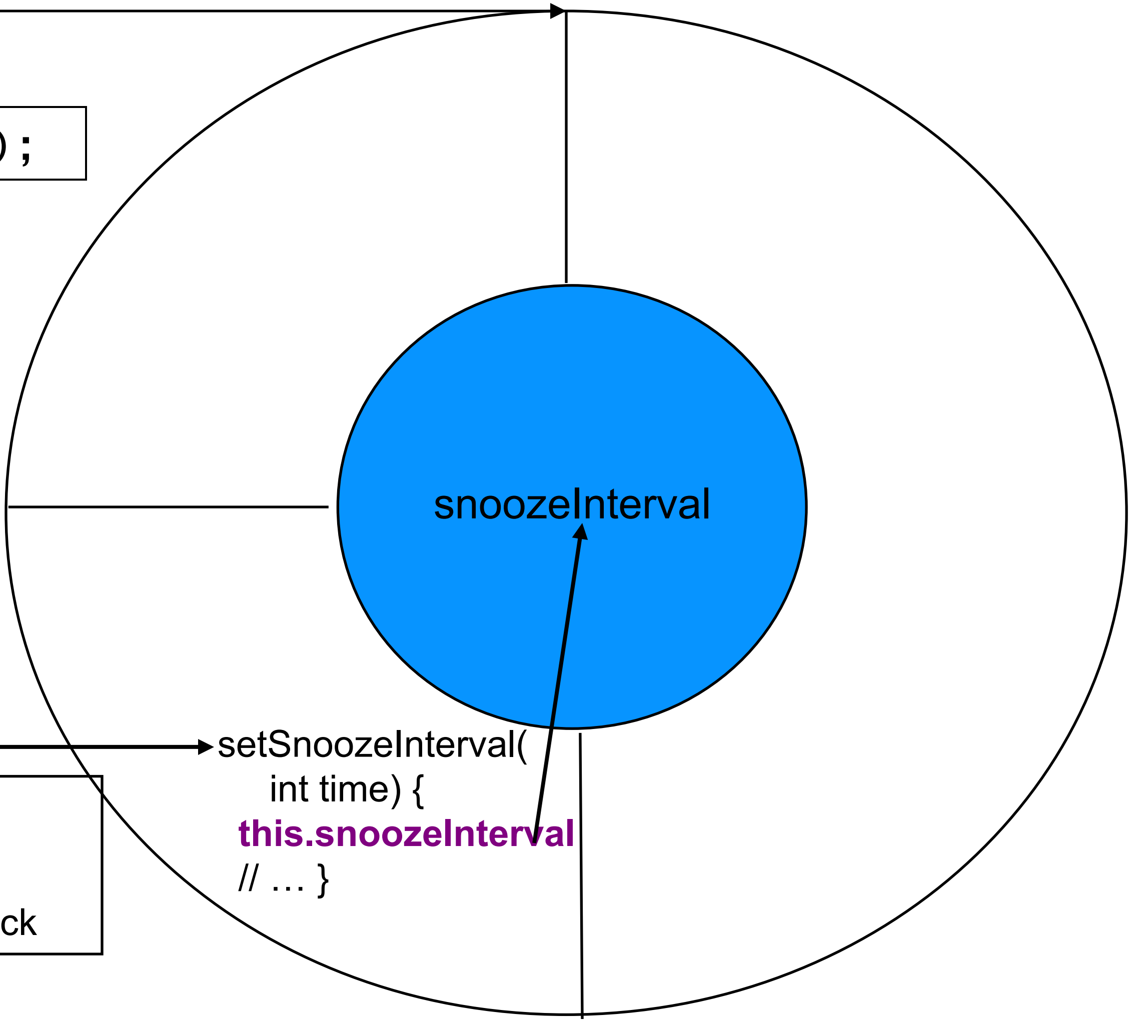
- **this**: A special variable referring to the object which invoked a method
  - It's available inside every instance method (see below)

```
public class AlarmClock {
  int snoozeInterval = 0;

  void setSnoozeInterval(int snoozeInterval) {
    // If the parameter and instance variable names are
    // the same, use this. to differentiate
    this.snoozeInterval = snoozeInterval;
  }

  int getSnoozeInterval() {
    return snoozeInterval;   // Returns the instance var
  }
}
```

Pretty Pictures

```
AlarmClock myClock = new AlarmClock();
```

```
myClock.setSnoozeInterval(100);
```



The this reference is initialized when invoking `setSnoozeInterval()`.

It references the same object as myClock

Overloading Methods

- Method names can be reused in a class (very handy)
  - e.g. `System.out.println(…)` has many variants
- Called ***method overloading***
  - The methods **must** have **different parameter lists**
  - They may (or may not) have different return types

```
// System.out is actually a PrintStream object
// much detail omitted
public class PrintStream {
  public void println(Object obj) { /* ... */ }
  public void println(String s)   { /* ... */ }
  public void println(int i)      { /* ... */ }
  public void println(long l)     { /* ... */ }
  // etc.
}
```

Calling Overloaded Methods

- The specific method called is based on **the arguments passed** in the call
  - Determined at compile time by choosing the closest match to the passed parameters
  - The **return type is not used** to resolve the call

```
public class PrintStreamExample
{
  public static void main(String[] args)
  {
    // call println(String)
    System.out.println("2 + 2 = ");
    // call println(int)
    System.out.println(4);
  }
}
```

The toString() Method

- **toString()**: A special method in Java
  - Returns a "String representation of the Object"
    - As decided by the writer of a class
  - The default returns a string concatenating the:
    - Class name
    - Return value of the hashCode() method [1]
  - You can override the default toString()

```
public class AlarmClock {
  int snoozeInterval = 0;

  // Here's a very simple version
  public String toString() {
    return "You're set to snooze for: " + snoozeInterval;
  }
}
```

Encapsulation and Access Control

Working with Methods and Data

Encapsulation and Access Control

Constructors

`static` or Class Members

Type-Safe Enums

Other Language Details

Encapsulation: Black Boxes

- Many terms for encapsulation
  - – Information Hiding
  - – Abstraction

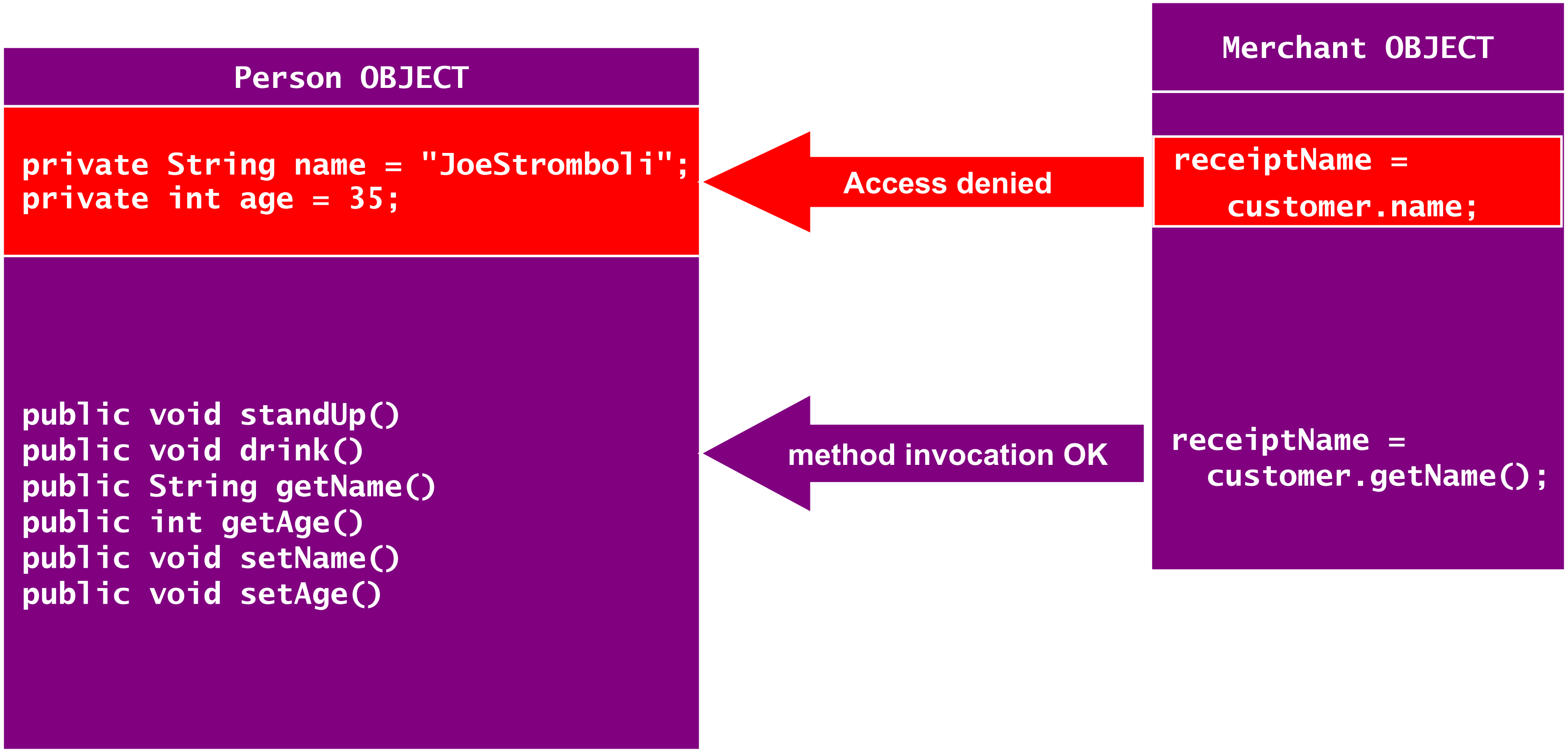
- Helps manage complexity
  - – Only reveal the essentials
  - – Reduces the details known externally

Encapsulation

- **Encapsulation**: "Black box" approach to building types
  - **Hides** a type's implementation
  - Core principle of OOP
  - Uses interact only through the **exposed behavior**
  - Internals **are hidden**, and never accessed directly
- Consider a `Person` type, with the behavior `standUp`
  - To have a person stand up, clients want to invoke the behavior
  - And not deal with a bunch of muscles and ligaments
- Consider a person's age
  - Can you set the age to -25 (negative 25)?
  - You could if you accessed the age field directly

Encapsulation

- Encapsulation encloses and protects data in an object
  - Also known as "data hiding"



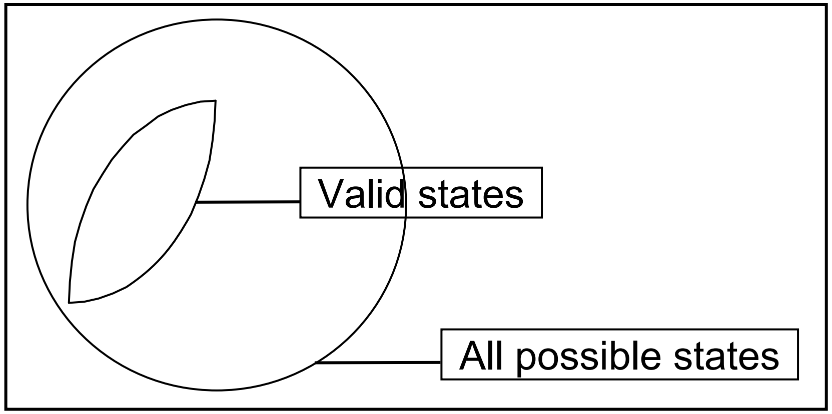
Key Advantages of Encapsulation

- Helps keep **programs correct**
  - Ensures data accuracy via data validation
  - Protects the data from external clients (private access only)
- **Isolates users from changes**
  - Reduces maintenance by localizing changes
  - Hides implementation details as well as the data
  - Improves flexibility and reusability
- **Abstracts detail**
  - Users interact at a higher abstraction level
  - They deal with a few high-level behaviors
    - And not many low-level details
  - End result - usage is greatly simplified

Encapsulation Helps Program Correctness

- Type generally have many possible states
  - Only some are valid
  - e.g. an `int` holding a person's age can have a negative value
  - But only positive ages make sense
  - Setting age via a method (only), can ensure valid values

- Accessing data through methods lets you **ensure validity**
  - You can't with direct access to fields



Access Control

- **Access protections** enforce encapsulation in Java
  - Via the `public`, `protected`, and `private` keywords
  - They let you control external access to both data fields and methods

- This access control is required for encapsulation
  - Expose only the methods that are appropriate for external use
    - e.g. make them `public`
  - Keep the implementation `private`
    - Including methods only suitable for internal use

Access for Data Members and Methods

- `public`, `protected`, and `private` appear directly on the field or method
  – The access modifier goes before the type name, e.g.

```
private int snoozeInterval = 0;
```

- Very often, **fields are kept `private`** while methods are made `public`

- We'll look at `public` and `private` now
  – We'll talk more about `protected` later
  – We'll also cover the default access level
    - When no level is specified

Private Access

- A **private** member is only directly accessible by code **within the class** where it's defined

```
public class AlarmClock {

  // Make the data private for encapsulation
  private int snoozeInterval = 0;

  void setSnoozeInterval(int snoozeIn) {
    snoozeInterval = snoozeIn;
  }
  int getSnoozeInterval() {
    return snoozeInterval;   // Returns the instance var
  }
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    myClock.snoozeInterval = 10; // ERROR: not accessible
  }
}
```

Public Access

- A **public** member is accessible to any other class

```
public class AlarmClock {

  private int snoozeInterval = 0;

 // Make the methods public for access
  public void setSnoozeInterval(int snoozeIn) {
    snoozeInterval = snoozeIn;
  }
  public int getSnoozeInterval() {
    return snoozeInterval;   // Returns the instance var
  }
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    myClock.setSnoozeInterval(10); // OK: Method accesible
  }
}
```

Lab 4.1: Encapsulation

In this lab, we will encapsulate the data in Television

Constructors

Working with Methods and Data

Encapsulation and Access Control

Constructors

`static` or Class Members

Type-Safe Enums

Other Language Details

Constructors

- **Constructors**: Special method called when creating an instance
  - Used to **initialize** the new object
  - They have the **same name as their class**
  - **No return value** is specified
    - **A reference to an instance of the class itself is the return value**
    - Since that's what's being created

```
public class AlarmClock {
  private int snoozeInterval = 0;

  // Create an AlarmClock with the default snooze time
  public AlarmClock() { }

  // Create an AlarmClock with the specified snooze time
  public AlarmClock(int snoozeIn) {
    setSnoozeInterval(snoozeIn);  // Why do it this way?
  }
  // …
}
```

Using Constructors

- Constructors are invoked when an object is created using **new**
  - A constructor is selected based on the arguments passed
  - As with overloaded methods

- **Note**: The ***no-argument*** or ***default*** constructor is special
  - i.e. - the constructor that takes no arguments
  - **If you declare no constructors**, the following constructor is **implicitly and automatically provided** by the Java compiler:

```
AlarmClock() { }
```

  - **If you declare any constructor**, this is **NOT** automatically declared
    - If you want a no-arg constructor, define it, as we saw in `AlarmClock`

```
// create AlarmClock with no-arg (or default) constructor
AlarmClock a1 = new AlarmClock();

// create AlarmClock with constructor that takes an int
AlarmClock a2 = new AlarmClock(1500);
```

Explicit Constructor Call

- One constructor can invoke another constructor
  - You must do it in the first statement
  - Invoke via **this()**, with a list of arguments in parentheses

```
public class AlarmClock {
  private int  snoozeInterval = 0;
  private long currentTime = 0;

  public AlarmClock(int snoozeIn) {
    // invoke other constructor explicitly
    this(snoozeIn, 0);
  }

  public AlarmClock(int snoozeIn, long currentIn) {
    setSnoozeInterval(snoozeIn);
    setCurrentTime(currentIn);
  }
}
```

Lab 4.2: Adding Constructors to a Class

In this lab, we will add constructors to `Television`

static or Class Members

Working with Methods and Data
Encapsulation and Access Control
Constructors
static or Class Members
Type-Safe Enums
Other Language Details

Static Fields and Methods

- A `static` or ***class*** field is associated with a particular class
  - Not with an instance as for a regular field (instance variable)
  - Often used to define class constants
- Only **one copy** of a static field exists
  - Allocated and initialized when a class is first loaded
  - It's independent of any instance
  - It's shared by all users of the field
- `static` methods can be invoked **without an instance**
  - Not associated with an instance
  - `static` methods do not have a reference to `this`
    - Since there is no instance associated with them
  - Often used for utility methods

Declaring Static Members

- Use the **static** keyword to define static fields and methods

- The example below comes from `java.lang.Math`
  - It shows both a static field and static method
  - The class holds many constants and utility methods

```
public final class Math
{
  public static final double PI = 3.14159265358979323846;
  public static double sin(double a) { /* … */ }
  // ...
}
```

Accessing Static Members

- Reference a static member through the **class** name, as shown below
  - **Not** through an instance
  - Static members are considered part of the class

- Below, `PI` is a static field of the `Math` class
  - `sin()` is a static method of the `Math` class

```
class UsingPI
{
  public static void main(String[] args)
  {
    System.out.println("Pi equals " + Math.PI);
    System.out.println("sin(0) =  " + Math.sin(0));

  }
}
```

Accessing Data in Static Methods

- Static methods can only access static fields

```
public final class Math {
  // this method isn't actually in the Math class,
  // but if it was, you could write it like this
  public static double circum(double radius) {
    return 2 * PI * radius;
  }
  public static final double PI = 3.14159265358979323846;
}
```

- **Static methods can't access instance variables**
  - There is **NO instance** to access them through

- Note that `circum()` can access `PI` without saying `Math.PI`
  - They are in the same class, and so in the same namespace

`final` Variables

- Use the **`final`** modifier to declare variables as constant (read-only)

```
public final class Math {
  public static final double PI = 3.14159265358979323846;
  ...
}
```

  - A final variable **must have an initializer**
  - It is a compiler error to modify a final variable
  - Final data members are **often** static
    - If they always have the same value, why keep multiple copies?
  - **public static final** variables are also called ***class constants***
    - They often use an **ALL_CAPS** naming convention

- In the example, refer to the variable as **`Math.PI`**

[Optional] Lab 4.3: Using `static` Members

In this (optional) lab, we will add static variables and methods to `Television`

Type-Safe Enums

Working with Methods and Data
Encapsulation and Access Control
Constructors
`static` or Class Members
Type-Safe Enums
Other Language Details

Enumerated Types Defined

- A type that has a known set of fixed values
  - Often used static constants in early Java versions (see below)
  - You can now use enums (next slide)

- Examples from everyday life:
  - **Gender**: male, female
  - **Month**: January, February, ..., November, December
  - **Day**: Sunday, Monday, ..., Friday, Saturday

```
class Gender {
  public static final int MALE = 0;
  public static final int FEMALE = 1;
  ...
}
// e.g. to specify a gender use Gender.FEMALE
```

enum Types

- A Java **enum** is a type-safe, robust enumerated type
  - You declare one by using the **enum** keyword
  - Below, we show some examples of how it might be used (1)
- They have many of the same properties as regular classes
  - Under the hood, an enum gets compiled into a class (2)

```
package org.darwin;

public enum Gender {MALE, FEMALE}
```

```
import org.darwin.Gender;
class GenderTest {
  public static void main(String[] args) {
    Person p = new Person("Robin", 16, Gender.FEMALE);
    Cat kitty = new Cat("Velvet", Gender.MALE);
    ...
```

More `enum` Examples

```
public enum Day {
  // Each value is an instance of Day
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}
```

```
These are referred to as Day.FRIDAY, Direction.WEST, etc.
```

```
// The below prints out the text FRIDAY
System.out.println(Day.FRIDAY);
```

```
// Each value is an instance of Direction
public enum Direction {NORTH, SOUTH, EAST, WEST}
```

- As with `int` class constants, the recommended naming convention is ALL_CAPS [(1)]

enum Features

- Printing out a value prints out the lexical equivalent

```
// The below prints out the text FRIDAY
System.out.println(Day.FRIDAY);
```

- Use **`ordinal()`** to get a numerical value (1)

```
// The below prints out the value 0
System.out.println(Day.SUNDAY.ordinal());
```

- Use **`valueOf()`** to convert a string to an enum value

```
// newDay will have a value of Day.SUNDAY
Day newDay = Day.valueOf("SUNDAY");
```

switch on enum

- You can write `switch` statements with enums
  - Formerly, you could only switch on `byte`, `short`, `int`, `char`

```
public void move(Direction where)
{
  switch (where)
  {
    case NORTH:
      ...
    case SOUTH:
      ...
    case EAST:
      ...
    case WEST:
      ...
  }
}
```

for-each with enum

- Every enum has a `static` **`values`** method [1]
  - Returns an array containing all of the enum's values
  - Very useful for iterating over an enum with for-each

```
for (Day d : Day.values())
{
  System.out.println(d);  // prints "SUNDAY", "MONDAY", etc
}
```

- **`EnumSet`** and **`EnumMap`** have been added to `java.util`
  - Special-purpose `Set` and `Map` implementations just for enums [2]
  - `EnumSet` has a `static` **`range`** method to return an enum subset

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
{
  System.out.println(d);  // weekdays only
}
```

Advanced enum Features

- An enum is really a class, so you can add data and methods

```
public enum Month {
  JANUARY  (31),  // calls constructor with 31
  FEBRUARY (28),  // calls Constructor with 28
  MARCH    (31),
  APRIL    (31),
  MAY      (31),
  JUNE     (30);  // Note the semicolon - required if you
                  // have additional declarations below
  private final int days;  // each Month knows its #days

  Month(int daysIn) {  // Constructor - cannot be public
    days = days;    // enum can't be instantiated directly
  }

  public int days() {
    return days;
  }
}
```

Problems with `int` Enumerated Types

- Prior to Enums (Java 5), Java used the `static int` pattern
  - Still used, but has many issues
- **Not type-safe**
  - Pass any `int` where a gender is expected (e.g. 5)
  - You can add two genders together (which means what?)
- **Brittle**
  - `static final` constants are compiled as simple values (1)
  - Adding a new enumeration value, requires recompiling all users
    - Otherwise, their behavior is undefined (2)
- **Uninformative value** for debugging and display
  - Displaying one yields only a number
  - It indicates nothing about its meaning

LEARNINGPATTERNS

Lab 4.4: Thinking About enums (Discussion-Only Lab)

In this lab, we will discuss a `Television` implementation that uses an enum for volume

Other Language Details

Working with Methods and Data

Encapsulation and Access Control

Constructors

`static` or Class Members

Type-Safe Enums

Other Language Details

Scopes and Blocks

- Blocks organize declarations and statements
  - **block**: A sequence of variable declarations and statements enclosed by curly braces **{ }**
  - We've seen these in classes and methods
  - A block defines a new scope

- A **scope** is a distinct context within a program
  - A scope introduces a new ***namespace***
    - A distinct context for names (e.g. for variable names)

Scope Example

- Fields and methods are accessible to the entire class
- Local variables and method parameters are local to their method

Class scope

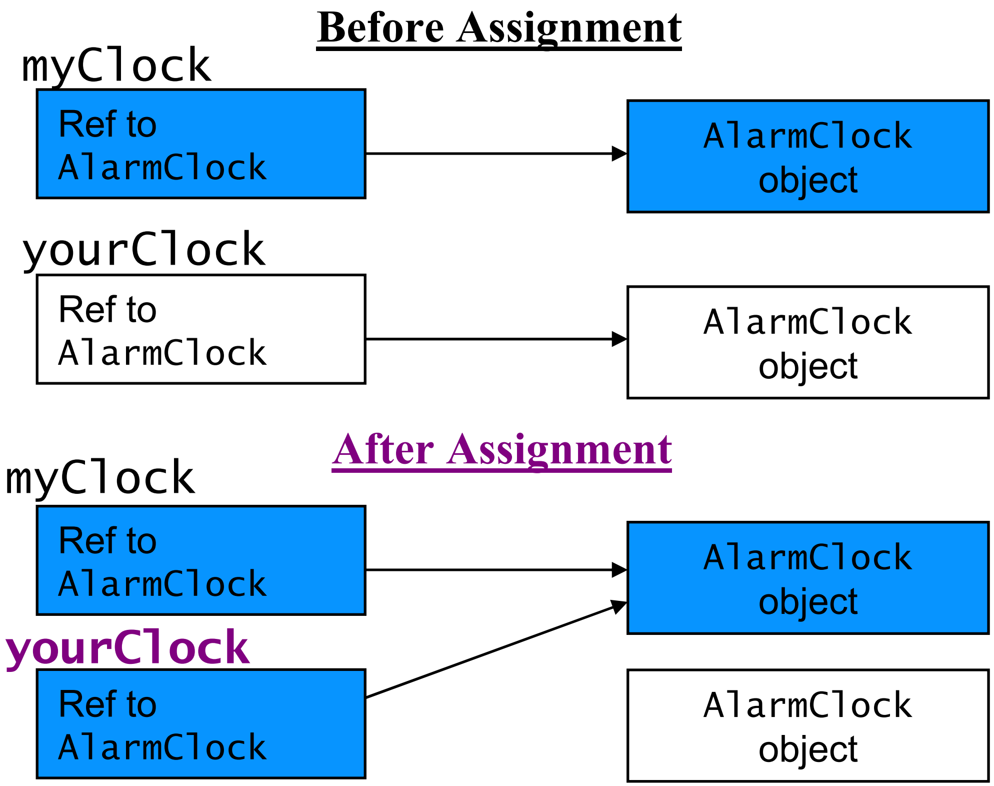
Method parameter and local variable scope

```
class ScopeSlide
{
  // member (instance) variable declarations
  int x = 56;
  // method definitions
  public void scopeMethod(int num)
  {
     int y = x + num;
     System.out.println("How many people?");
     System.out.println("This many: " + y);
  }
}
```

Assignment

- Assigning to an object reference variable just changes what it refers to

```
AlarmClock myClock = new AlarmClock();
AlarmClock yourClock = new AlarmClock();
yourClock = myClock;  // yourClock ref to same object as myClock
```



Comparison

- You can compare object references with **==** and **!=**
  – To check if the references refer to the **same instance**

```
AlarmClock myClock = new AlarmClock();
AlarmClock yourClock = myClock;

if (myClock == yourClock) {  // true in this case
   System.out.println("one Clock");
}
```

- **== tests for *identity***
  – i.e., checks if the two references refer to the **same object**
  – We'll see later how to test for ***equality*** (two objects having the **same value**)

Null Objects

- `null`: A special object reference meaning "not a valid object"
  - Can be assigned to any class type variable

```
// Initialize myClock to point to null
AlarmClock myClock = null;
```

- Test objects for `null` using `==` and `!=`

```
if (myClock == null)   // true
{
  System.out.println("Oh My! We're out of time!");
}
```

Wrapper Classes

- Primitive types are not objects
  - There is no class associated with them
  - Done for efficiency
  - All other data are objects of some class type

- ***Wrapper classes*** represent primitive types as objects
  - e.g. class **`Integer`**: A class wrapper around a primitive **`int`**

- Provide functionality not in the primitive types
  - Data conversion from a string value
  - Let you treat primitive types as objects when needed
  - The notes have more details on this

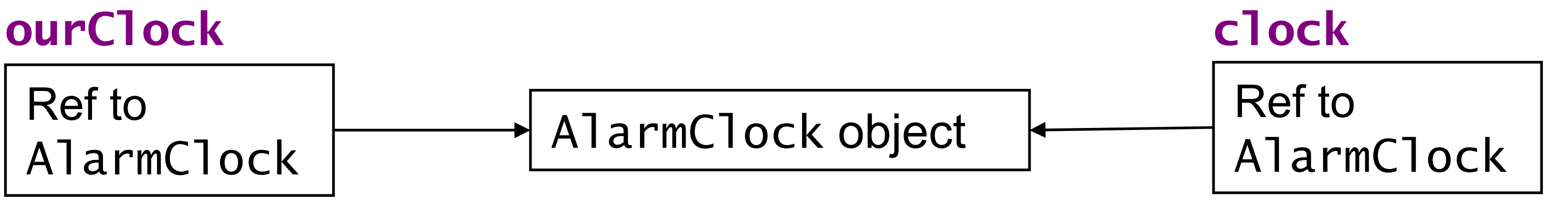
Reference Types as Method Parameters

- **Class type** method parameters are passed by **reference**
  - **The object is not copied - just the reference**
- Consider `Worker` below
  - `sleepMore()` receives an `AlarmClock` reference
  - The actual instance is created elsewhere
  - Within `sleepMore`'s scope the reference "`clock`" is another reference for this instance

```
public class Worker {
  void sleepMore(AlarmClock clock) {
    // The Worker's family will be surprised!
    // Everyone using the clock will be snoozing
    clock.setSnoozeInterval(5000);
  }
}
```

Reference Types as Method Parameters

- Below, the worker is passed a clock when calling `sleepMore()`
  - It's now referred to by "`ourClock`" and "`clock`"
  - Anything done through "clock" in `sleepMore()` is seen by any other code referencing this instance



`Inside a main method`

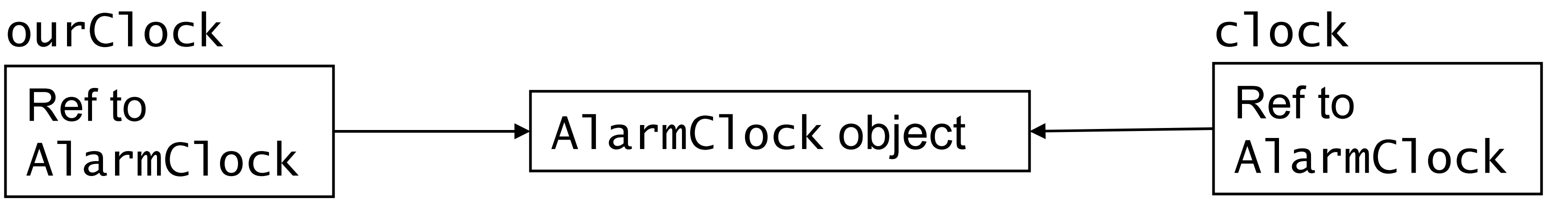
```
// Clock with snooze of 1500
AlarmClock ourClock =
    new AlarmClock(1500);
Worker w = new Worker();
w.sleepMore(ourClock);
int snooze =
  ourClock.getSnoozeInterval();
// what is snooze? (5000!)
```

`Worker`

```
void sleepMore(AlarmClock clock) {
  // The Family will be surprised!
  clock.setSnoozeInterval(5000);
}
```

final Method Parameters

- For method parameters that are marked **final**
  - Primitive/value types: **the value can't be changed**
  - Class/reference types: **the reference can't be changed**
    - **But the object's data still can(!)**



Family

```
AlarmClock ourClock =
    new AlarmClock(1500);
Worker w = new Worker();
w.sleepMore(ourClock);
int snooze =
  ourClock.getSnoozeInterval();
// what is snooze? (5000!)
```

Worker

```
void sleepMore(final
               AlarmClock clock) {
  // Can modify clock's data
  clock.setSnoozeInterval(5000);
  // cannot modify clock ref
  clock = null;  // ERROR
}
```

[Optional] Lab 4.5: Debugging

In this lab, we will learn about and use the debugging capabilities of the IDE

Review Questions

1. What is method overloading? What are the programming implications of calling overloaded methods?
2. What is encapsulation, why is it useful and how is it enforced?
3. What are constructors? Why are they useful?
4. What does the == operator check for when used with two objects?
5. What is an enum, and what is it used for
6. What are `static` (class) variables and methods?
7. Why can't `static` methods call the instance methods present in the same class?

Lesson Summary

- **Overloaded** methods let you give **methods the same name**
  - With different signatures
  - Convenient for multiple methods providing the same behavior

- **Encapsulation hides a type's implementation**
  - Clients interact with behavior only (methods)
  - Isolates them from implementation changes, abstracts detail, and helps keep data valid
  - Supported in Java via the **`private`** keyword

- **Constructors** are methods **called when an object is created**
  - Supports object initialization
  - Have the same name as the class, and no return value

Lesson Summary

- The **==** operator checks for **identity**
  - Do you have the exact same object?
- An **enum** is a type-safe enumerated type
  - Declared with the **`enum`** keyword, and a list of allowed values
- A **static** (or class) field or method is **associated with a class**
  - Not an instance
  - static fields often define class constants
    - **`public static final`** variables
  - Static methods can be **invoked without an instance**
    - There is no instance associated with a static method
    - **They can't call instance methods or access instance variables**

Session 5: Flow of Control

Branching Statements

Iteration Statements

Session Objectives

- Outline the comparison and boolean operators in Java
- Discuss branching statements and the operators used with them
  - `if`, `if-else`, `switch`
- Discuss iteration (looping) statements
  - `while`, `do-while`, `for`
  - `break`, `continue`
- Use flow of control logic to perform data validation in an object

Branching Statements

Branching Statements

Iteration Statements

Program Execution Sequence in Java

- Statements in Java are executed in sequence
  - Unless otherwise directed
- Java has fairly standard flow control statements
  - Branching/selection statements choose one of several flows:

    `if`, `if-else`, and `switch`
  - Iteration statements specify looping

    `while`, `do-while`, and `for`
  - Jump statements transfer control unconditionally

    `break`, `continue`, `return`

The Comparison Operators

- Selection and iteration statements use comparisons producing a boolean (logical) result
  - `true` or `false`
- The operators below compare numerical values and produce `boolean` results

Operator		Example
`==`	equal	`3 == 5 (== false)`
`!=`	not equal	`3 != 5 (== true)`
`<`	less than	`3 < 5 (== true)`
`>`	greater than	`3 > 5 (== false)`
`<=`	less than or equal	`5 <= 5 (== true)`
`>=`	greater than or equal	`5 >= 5 (== true)`

The Logical Operators

- Compare `boolean` values and produce `boolean` results.

- In the table below, we assume:

```
boolean t = true;
boolean f = false;
```

Operator	Meaning	Example
&	AND	`f & t` (== `false`, `t` evaluated)
&&	conditional AND	`f && t` (== `false`, `t` not evaluated)
|	OR	`t	f` (== `true`, `f` evaluated)
||	conditional OR	`t || f` (== `true`, `f` not evaluated)
^	exclusive OR	`t ^ t` (== `false`) `t ^ f` or `f ^ t` (== `true`)
!	NOT	`!t` (== `false`)
==	equal	`f == f` (== `true`)
!=	not equal	`t != t` (== `false`)

`if-else` Statement

- **`if`** / **`if-else`** statements control statement execution by the value of an expression
  - `if` (required) is executed for `true` values
  - `else` (optional) is executed for `false` values

`if (` *Expression* **`)`** *Statement* **`else`** *Statement*

```
int i = 1;

if (i == 0) {
  System.out.println("i equals 0");
} else {
  System.out.println("i is not 0");
}
```

`switch` Statement

- **`switch`** allows testing for more than one value
  - Can switch on `byte`, `short`, `int`, `char`, `String` and `enum`
  - `default` case is optional and gets control when no case matches

- Note: If the **break** is **not present**, **execution continues on** to the code in the next case

```
int i = 1; // This would usually be initialized elsewhere
switch ( i ) { // Execute a case based on value of i
  case 1:
    System.out.println("i is 1");
    break;
  case 2:
    System.out.println("i is 2");
    break;
  default:
    System.out.println("i is large");
    break;
}
```

Iteration Statements

Branching Statement

Iteration Statements

while Statement

- **while** creates a loop – it has the form:

while (*Expression* **)** *Statement*

```
int index = 10;

while (index > 0) // stop looping when index reaches 0
{
  // do some work
  index = index - 1;
}
```

```
ResultSet rs = ...; // This is a JDBC ResultSet object
while (rs.next())  // rs.next() eventually returns false
{
  // process next row in result set
}
```

do-while Statement

- `do-while` creates a loop – it has the form:

do *Statement* **while (** *Expression* **)**;

```
int index = 10;

do
{
  // do some work
  index = index - 1;
}
while (index > 0); // stop looping when index reaches 0
```

- What's the difference between `while` and `do-while`?
  - What happens if the index is initially set to 0, before the do-while is executed?

for Statement

- **for** creates a loop – it has the form:

for (*Initialization*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Increment*$_{opt}$ **)** *Statement*

```
// Print values from 0 to 3
for (int i = 0; i <= 3; i++)
{
  System.out.print(i + " ");
}
```

- There is also a `for-each` loop that we'll see later

break Statement

- **break** transfers control to the end of the enclosing loop (`for`, `while`, `do-while`) or `switch` statement
- It has the form:

break;

```
// this method scans values looking for a specific one
void findValue(int value)
{
  for (int i = 0; ; i++) // No terminate expression
  {
    if (i == value)
    {
      System.out.println("got the value");
      break;  // stop looping, found the value
    }
  }
  // control is here after the break
}
```

continue Statement

- **continue** exits the **current iteration** of the loop (`for`, `while`, `do-while`)
  - It continues with the next iteration
- It has the form:

continue;

```
// here we use continue to print out even numbers only
for (int i = 1; i <= 10; i++)
{
  if ((i % 2) != 0)  // if not divisible by 2 (not even)
    continue;        // exit this iteration of the loop

  System.out.println(i);
}
```

Lab 5.1: Data Validation

In this lab, we will add data validation to a class

Review Questions

1. Name the two AND operators in Java. What is the difference between them?
2. True or false: `if` and `if-else` statements require the use of blocks
3. What data types can you use to control a `switch` statement?
4. What is the difference between `while` and `do-while`?
5. What is the difference between `break` and `continue`?

Lesson Summary

- Java contains two versions of the AND and OR operators
  - **`&&`** (conditional AND) and **`&`** (normal AND)
  - **`||`** (conditional OR) and **`|`** (normal OR)
- Conditional and looping statements in Java can work on single statements or blocks enclosed with `{}`
- `switch` operate on integer values, enums, and strings
- Java supports a number of looping statements:
  - **`while`**: may not execute loop body
  - **`do-while`**: always executes body at least once
  - **`for`**: numeric looping
  - You can modify loop flow via
    - **`break`**: Break out of a loop (or switch)
    - **`continue`**: exit current iteration, and go to next iteration of loop

LEARNINGPATTERNS

Session 6: Strings and Arrays

`String`, `StringBuffer`, and `StringBuilder`

Arrays

Session Objectives

- Extend your knowledge of how `String` objects work
- Know when to use `StringBuffer`/`StringBuilder`
- Understand the characteristics of arrays and how to use them in your code

LEARNINGPATTERNS

String and StringBuffer/StringBuilder

String, StringBuffer, and StringBuilder

Arrays

Using Strings

- `String` objects and string literals (e.g. "Hi") are special
  - A `String` object may be initialized by a string literal

```
String s = "Hello World";
```

  - Creates a new `String` instance initialized to `"Hello World"`
    - Sets s to refer to it
  - But **you never use new**

- Concatenate strings using **+**

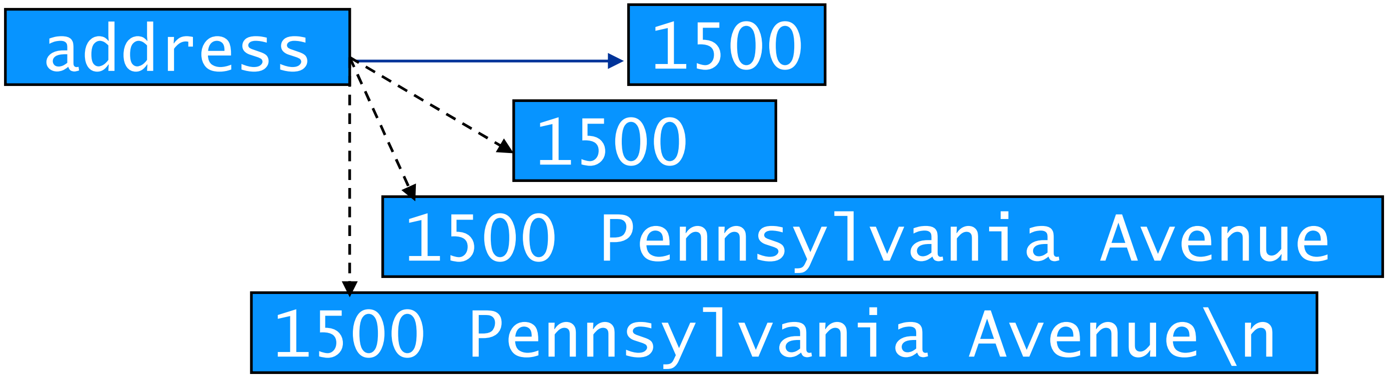
```
String s = "Hello World";
// create String containing "Hello World, how are you?"
String t = s + ", how are you?";
```

  - Note: This is the only support for using built-in operators (e.g. +) on objects

Changing Strings

- `String`s instances are immutable - you can't change their contents (1)
  - But you can point a reference to a new string
    - The old instance is no longer reachable by that reference
  - Below, we create new string instances with each concatenation

```
String address = "1500";
address = address + " ";
address = address + "Pennsylvania Avenue";
address = address + "\n";
```



Classes `StringBuffer` and `StringBuilder`

- **`StringBuffer/StringBuilder`**: Preferred way to create / manipulate strings
  - They have the same API
  - `StringBuffer` is synchronized/thread-safe
  - `StringBuilder` is not, and performs faster

- Common methods include:
  - `public` **`StringBuffer`**`()`: Construct a new empty `StringBuffer`
  - `public` **`StringBuffer`**`(int capacity)`: Construct a new `StringBuffer` with the given capacity
  - `public StringBuffer` **`append`**`(String str)`: Append `str` to the current `StringBuffer`
    - Other methods append primitive types and `char` arrays

StringBuffer and StringBuilder

- Represent a string instance modifiable using no additional memory
  - Not immutable - methods modify an internal string buffer
  - Call `toString()` to extract the value as an (immutable) `String`
  - As shown below

```
StringBuffer address = new StringBuffer(50);
address.append("1500");
address.append(" ");
address.append("Pennsylvania Avenue");
address.append("\n");
String finalAddress = address.toString();
```

Regular Expressions

- Java supports regular expressions
  - Useful to validate a string's format
  - **`String.matches()`** check's a string's value against a regular expression:

    public boolean **matches(String regex)**

```
String ssn =     "077-23-0812";
String zip =     "10988-1223";
String letters = "aeYwdZQi";

// these all return booleans - usable in if statements, etc.
ssn.matches    ("[0-9]{3}-[0-9]{2}-[0-9]{4}");
zip.matches    ("[0-9]{5}(-[0-9]{4})?");
letters.matches("[a-zA-Z]*");
```

- See the `java.util.regex.Pattern` JavaDoc
  - Has detailed info on the pattern matching

Arrays

`String`, `StringBuffer`, and `StringBuilder`

Arrays

Arrays

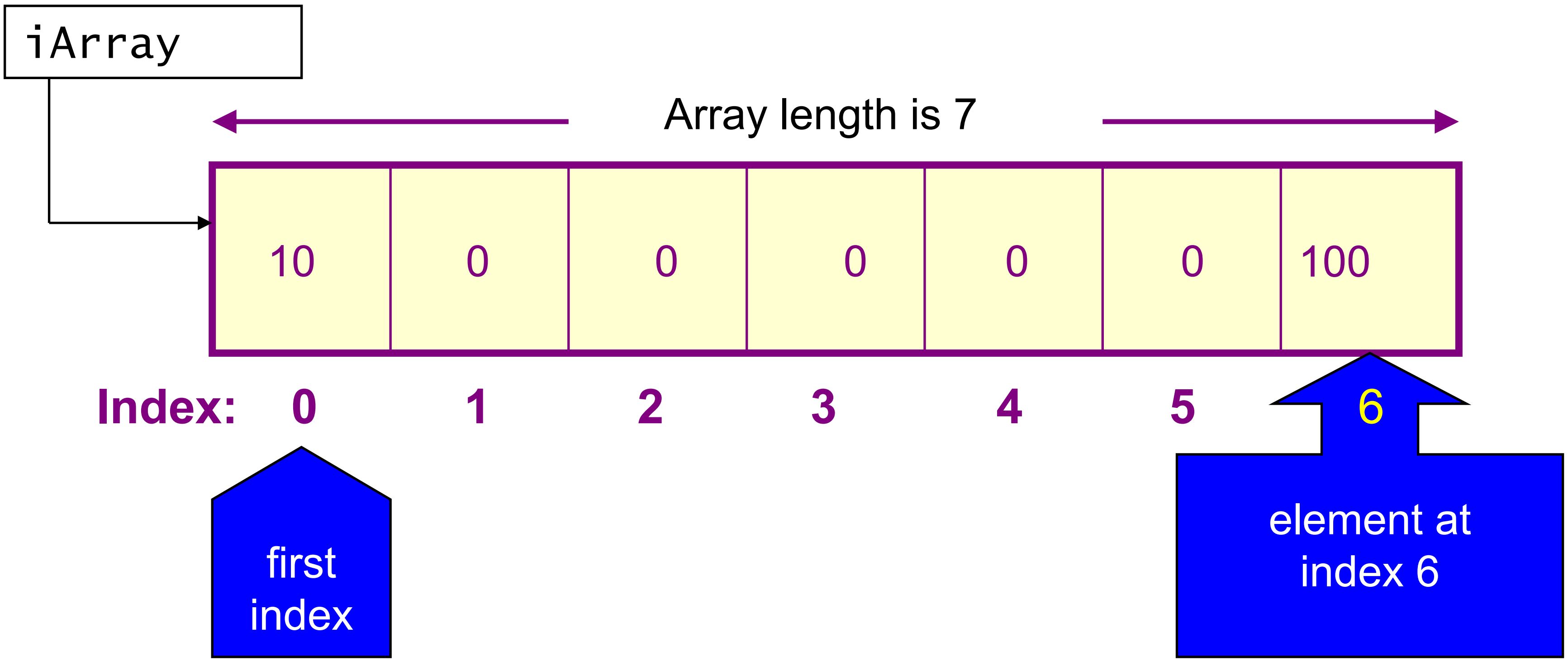
- Arrays hold collections of other data elements
  - You use **[]** to signify an array
  - You use **new** to create the array
- Used when the maximum number of items is **easily determined** and won't change
- **length**: Read-only field holding length of array
- When you declare an array reference, you must initialize it
  - It's an object, so acts like any other [1]
  - As shown below

```
int[] iArray1;       // reference to array of ints
int   iArray2[];     // alternate syntax

// int array reference initialized to array of 10 ints
iArray1 = new int[10];
int len = iArray.length;   // len == 10
```

Arrays

- Array indexing starts at **0**

```
int[] iArray = new int[7];
iArray[0] = 10;
iArray[6] = 100;
```



Creating Arrays and Accessing Elements

- Array creation and initialization

```
int[] tenPrimes = new int[10];
tenPrimes[0] = 2;
tenPrimes[1] = 3; // etc.
```

- Shortcut notation: Create an array object and fill it with values

```
int[] tenPrimes = {2,3,5,7,11,13,17,19,23,29};
int firstPrime = tenPrimes[0];    // == 2
```

- Use `[index]` to access individual array elements
  - Index ranges from 0 through array length - 1

```
System.out.println(tenPrimes[0]); // Print out the value 2
System.out.println(tenPrimes[10]); // ERROR - Exception thrown
```

- Using `length` - the read-only variable holding the size of an array
  - Note - you can't grow an array - you must copy it to grow it (1)

```
int[] tenPrimes = {2,3,5,7,11,13,17,19,23,29};
int arrayLength = tenPrimes.length   // == 10
```

Arrays of Object References

- An array can hold object references
  - The instances must be explicitly instantiated

```
// array of ten references to AlarmClock objects
AlarmClock[] clockArray = new AlarmClock[10];
clockArray[0] = new AlarmClock();   // initialize 1st clock
```

  - All array references are initialized to `null` when the array is created

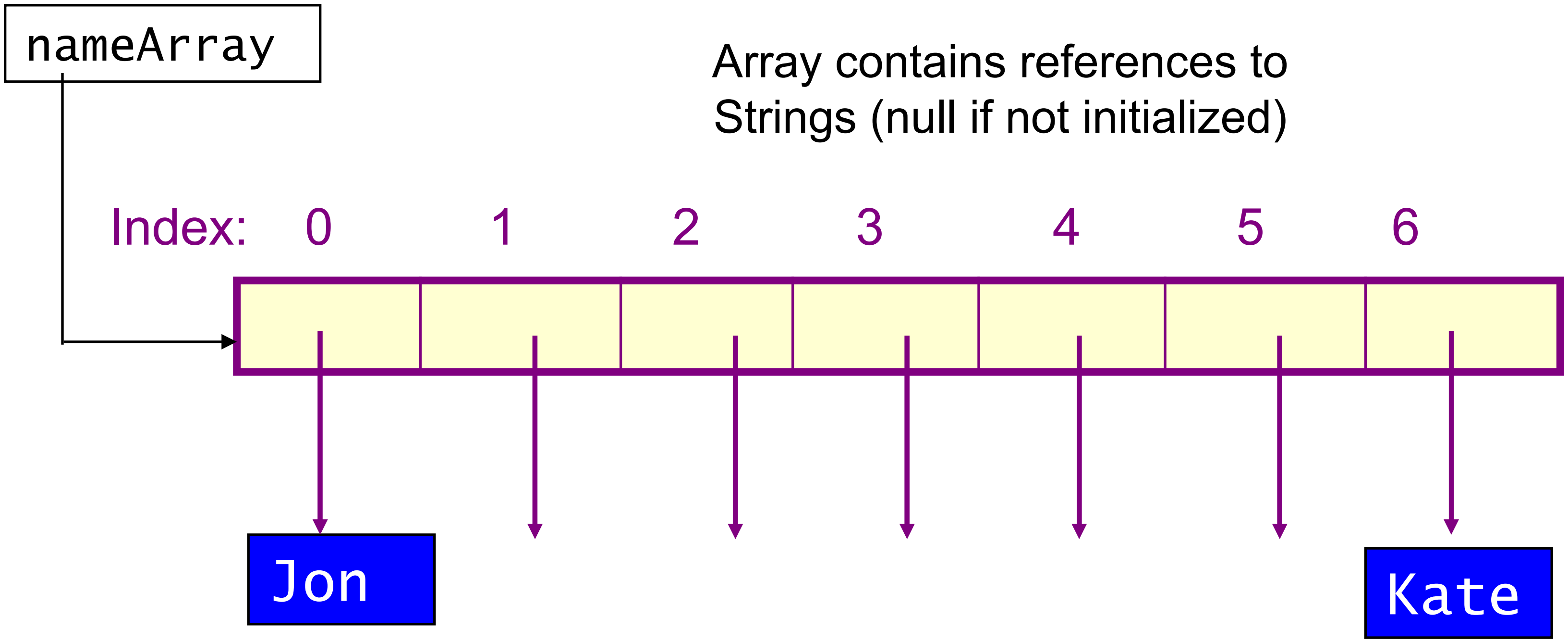
- The shorthand notation works for object instances
  - Below, we create an array and initialize it with two instances

```
AlarmClock[] clockArray = {new AlarmClock(),
                           new AlarmClock(100) };
```

Array of Strings

- The code below creates an array which we diagram at bottom

```
String[] nameArray = new String[7];
nameArray[0] = "Jon";
nameArray[6] = "Kate";
```



args Array

- `main()`'s arguments are passed in from the command line
  - Via the parameter **`String[] args`** (a String array)
  - Convert numerical arguments via the wrapper classes
  - String arguments with spaces must be enclosed in quotes

```
java TelevisionTest Hitachi RCA "My Brand"
```

- `main` should test for the existence and validity of its arguments

```
public static void main(String[] args) {
  if (args.length < 3) {
    System.out.println(
          "Usage: java TelevisionTest  Brand1 Brand2 Brand3");
  }
}
```

Iterating Over Arrays

- The **for-each** loop goes thorough all an array's elements
  - It's the preferred way to do this
  - Read the loop below as "**For each String s in args**

```
public static void main(String[] args) {
   for (String s : args) {
      System.out.println(s);
   }
}
```

- Can be done as shown below (1)
  - No longer common - more cumbersome than for-each

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

varargs

- **varargs** declare a method with any number of parameters
  - Via an ellipsis (three dots …) following an argument's type
  - The argument is treated as an array in the method
  - Only a method's **last parameter** can be varargs
- Below, `varMethod` takes any number of strings as an argument
  - `someMethod()` calls it with 1, 2, and no arguments

```
public void someMethod() {
  varMethod("Hi");
  varMethod("Hi", "Bye");
  varMethod();
}
private void varMethod(String... strings) {
  for (String cur : strings) { // Iterate over it
    System.out.println(cur);
  }
  System.out.println(strings.length); // Treat like array
}
```

Lab 6.1: Arrays

In this lab, we will practice using arrays

Review Questions

1. What is the difference between `String` and `StringBuffer`/`StringBuilder`?

2. How do you determine the number of elements in an array?

3. True or false: Java supports variable-length arrays.

Lesson Summary

- **`StringBuffer`**/**`StringBuilder`** let you manipulate strings efficiently
  - Not immutable, as native strings are
  - `StringBuilder` is the most efficient (no locking)

- Arrays in Java are **fixed length** collections of objects or values
  - Use square brackets **`[]`** to declare and work with the array
  - The read-only **`length`** field provides the current array size

Session 7: Packages

Packages Overview

`import` Statement

Creating Packages

Finding Classes

Session Objectives

- Explain some of the issues that packages address
- Understand the relationship between a package name and the directory structure on the filesystem
- Understand and use the `import` statement
- Create packages and put classes in them
- Understand what the classpath is and how it's used

Packages Overview

Packages Overview

`import` Statement

Creating Packages

Finding Classes

Dealing with Complexity

- So many classes, so little time
  - Systems often can have hundreds of classes (or more)
  - How do you organize them?

- Often, there are multiple classes with the same name
  - With slightly different functionality, or written by different groups
  - Java APIs have this, e.g.
    - A general purpose `Date` class
    - A `Date` class specialized for SQL

- **Packages** are used to solve these issues in Java

Packages

- ***package***: A collection of related declarations
  - Classes, interfaces, exceptions, enumerations …
  - For organization, access protection and namespace management
    - Classes are easier to find and use
    - Provides flexibility in access control
    - Helps avoid naming conflicts
- Package names are hierarchical
  - **Dot-separated list** of identifiers
- The Java API is organized in packages
  - Think of it as several libraries
  - Where the library name becomes part of the class name

package Statement

- **package** statements in source files identify packages
  – They appear at the top of the source file
  – Anything defined in the file is part of the package

```
// file: Date.java - in the java.util package
package java.util;

// public means it can be accessed outside the package
public class Date {
  // Date represents a specific instant in time
}
```

- The package name is part of the class name
  – **`java.util.Date`** is the full name of the class above

The Default (or Unnamed) Package

- Classes not in a named package are in the ***default package***
  - It has no name
  - Only for temporary code or very small programs
  - All classes in the same directory, without package statements, are in the default package for that directory (1)
  - In general, code should be in a named package

- Lets first look at how to use packages
  - We'll then look at how to create them

`import` Statement

Packages Overview

`import` Statement

Creating Packages

Finding Classes

The `import` Statement

- Imports go after the package statement, before anything else
  - They describe how to resolve the type names in your file
  - e.g. a class name like `Date`
- Two formats:
  1. Import all members of a package:

```
import packageName.*; // import java.util.*
```

  2. Import one package member

```
import packageName.MemberName; // import java.util.Date
```

- Three choices in using a package member (e.g. a class name)
  - Import all members in the package (1 above)
  - Import the specific class or interface (2 above)
  - Use the **fully qualified name** (`packageName.MemberName`) every time you use the member- (e.g. `java.util.Date`)

Importing a Complete Package

- To import **all the members** of a package:

  `import packageName.*;`

```
import java.util.*;
class AlarmClock {
   // Date is in package java.util
   Date currentTime = new Date();
   Timer snoozeInterval = new Timer();
   ...
```

- Why do it this way?
  – East of use – especially if using many members from a package

- Why not do it this way?
  – Doesn't document the classes used in this program
  – Can lead to ambiguity in what type is imported

Importing a Single Package Member

- To import **a single member** of a package:

  import packageName.Identifier;

```
import java.util.Date;
import java.util.Timer;
class AlarmClock {
  Date currentTime = new Date();
  Timer snoozeInterval = new Timer();
  ...
```

- Why do it this way?
  - Helps avoid naming conflicts
  - Documents/clarifies the member classes used
  - You may be using one or only a few members of a package
- Why not do it this way?
  - Quite a bit of typing, somewhat verbose
  - But development environments can help :-)

Using the Fully Qualified Name

- If not importing, use the fully qualified name every time

```
// e.g. To use the Date class in java.util
java.util.Date d = new java.util.Date();
```

- Why do it this way?
  - Sometimes needed to avoid naming conflicts
  - To be explicit for clarity
  - If you are using the member only once
  - Generated code often uses this style
- Why not do it this way?
  - Hard to maintain - if you need to change packages it can lead to a lot of changes
  - Lots of typing

Standard Imports

- Java automatically imports two packages in every file

1. The **current package**
   - Identified by the `package` statement in the file
   - Includes **all members** of the package your class is in
   - If there is no `package` statement, the current package is the default package of the current directory
   - That's how classes in earlier labs accessed each other

2. The **`java.lang` package**
   - A special package containing core Java types
   - An implicit `import java.lang.*` is done for you
   - Contains classes such as `String` and `System`
   - That's how earlier labs used them without any import
   - `String` is really `java.lang.String`

Resolving Naming Conflicts

- If members in different packages have the same name, you can have a naming conflict
  - For example `java.util.Date` and `java.sql.Date`
  - Just use the fully-qualified name

```
import java.util.*;  // Contains java.util.Date
import java.sql.*;   // Contains java.sql.Date
class AlarmClock {
  // compile ERROR: which Date?
  Date currentTime = new Date();
}
```

```
import java.util.*;
import java.sql.*;
// this is OK, because of fully qualified name
class AlarmClock {
  java.util.Date currentTime = new java.util.Date();
}
```

Creating Packages

Packages Overview

`import` Statement

Creating Packages

Finding Classes

Creating a Package

- Add a **package** statement to each member file of the package
  - Package names are hierarchical, with the form:
    `package identifier[.identifier…];`
  - Package members are generally functionally related
- Below, are two classes in the package `com.mycompany.time`
  - Packages generally named as reverse URLs (prevent collisions)

```
// File Timepiece.java
package com.mycompany.time;
import java.util.*;
class TimePiece { ... }
```

```
// File AlarmClock.java
package com.mycompany.time;
import java.util.*;
class AlarmClock extends TimePiece { ... }
```

Access Control for Class Members

- You can control access for class members
- **`public`** member: Accessible anywhere the defining class is accessible
  - If you can access the class, you can get to its `public` members
  - Constructors and many methods are often public
- **`private`** member: Accessible only in the defining class
  - Fields (data) are usually private
- Member with no declared access have "default access"
  - Also called "package private" access
- Note: You still need an instance to call regular methods
  - Even if you've imported their package

Access Control for Classes

- Classes must be `public` to be accessed from other packages

```
public class Date { … }
```

- **Only one** class per source file can be declared **`public`**

- The source file for a `public` class **must** match the class name, with a *`.java`* extension
  - `public class Date` **must** be defined in *Date.java*

- Generally, only one class is defined in a source code file
  - You can only have one top level public class
  - More advanced techniques might define other classes
    - e.g. inner classes, beyond the scope of this course

Summary - Using Packages

- **Add package statements** to your source files
  - The scope of a `package` statement is the entire source file
  - Every type defined in the file is in that package
  - Packages usually span multiple source files

- **Include access modifiers** (public/private) in your class definition
  - Fields (data) are generally private
  - Methods (behavior) are often public

- **Include `import` statements**, as needed, to use types outside your package

Finding Classes

Packages Overview

`import` Statement

Creating Packages

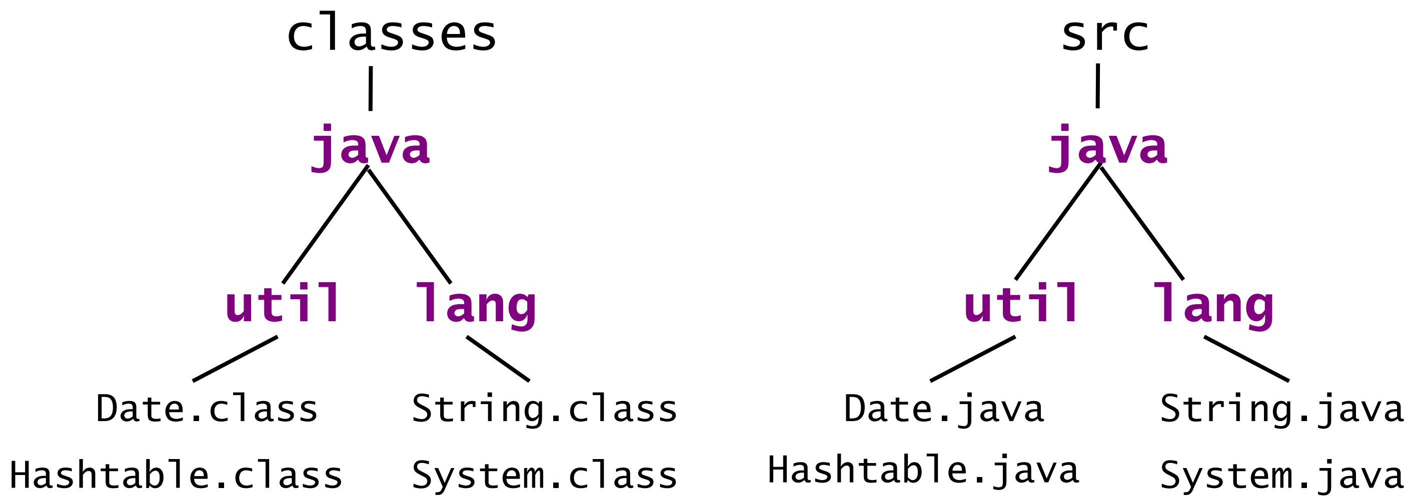
Finding Classes

Tools Must Locate Type Definitions

- To compile a source file requires knowing about the types it uses
  - The compiler looks at both .java and .class files to learn this
  - The compiler may compile other dependent files (as needed)
- To run an application, the JVM also needs all the types used
- How does the compiler and JVM find your application types?
  - They know how to get to the standard Java types
  - Let's look at the application types

Organizing Files and Packages

- Java organizes package files on the file system
  - All package files must be **in a subdirectory matching the full package name**
  - The directory names mirror the package names
    - e.g. **`java.util`** package files appear in a ***java\util*** directory
  - Illustrated below using two directories for .java and .class files
  - Top level package directories can be in any directory **on the classpath**



Classpath

- The classpath indicates where **Java tools look for types**
  – It specifies the **root locations** that are searched for types
  – Can be declared in the **CLASSPATH environment variable**
  – Can be given to tools using the **–classpath option**

- The classpath can contain **directories**, **zip**, and **jar files**
  – Entries are separated by a colon(*Nix) or semicolon(Windows)
  – Package subdirectories **can be on any directory on the classpath**

- **Zip and jar files** on the classpath are read by the tools
  – All of their contents are added to the classpath

Classpath Example

- There are three things on the classpath in the example below
  - The **current directory** - denoted by . (dot) (as well as any package subdirectories)
  - All classes in the ***myutil.jar*** file in the directory *c:\MyApp\lib*
  - The **directory** *C:\MyApp\classes*, and any package subdirectories
  - `MyApp`, at the end of the example, is the program to run, not part of the classpath

```
java -classpath .;c:\MyApp\lib\myutil.jar;C:\MyApp\classes MyApp
```

Classpath Example

- Assume you are running the program `TelevisionTest`
  - With all classes in the package `com.entertainment`
  - And all class files under *C:\StudentWork\workspace\Lab03.3*
  - The following command runs it (1)

```
java -classpath C:\StudentWork\workspace\Lab03.3
                        com.entertainment.TelevisionTest
```

  - The top level package directory (*com*) is under a directory on the classpath (*Lab03.3*)
  - The JVM will find the classes

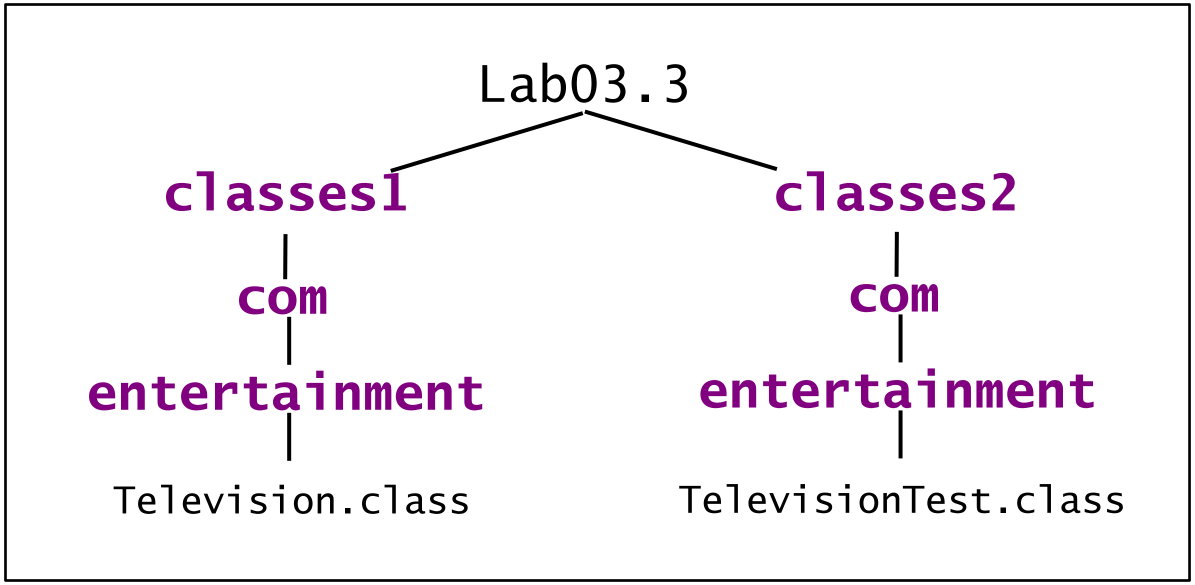
```
Lab03.3
   |
  com
   |
entertainment
   |
Television.class
TelevisionTest.class
```

Classpath Example

- Assume the classes were split into two directories as shown below
  - Run `TelevisionTest` as follows:

```
java -classpath
        C:\StudentWork\workspace\Lab03.3\classes1;
        C:\StudentWork\workspace\Lab03.3\classes2
        com.entertainment.TelevisionTest
```

  - The classpath contains both directories that hold our packages



What is a JAR?

- JAR stands for **J**ava **Ar**chive
  – It aggregates (collects) many files into one
- Based on the ZIP format
  – Standard packaging to distribute Java code (.class files, other jars, etc.)
  – It is **platform independent**
  – Can include resources, (audio, image, etc.) files
  – Supports authentication via digital signatures
- Includes a manifest file that
  – Lists the filenames in the jar
  – Specifies algorithms to compress and/or sign the files
  – Other information - see the Jar file spec for more info

JAR Classpath Example

- Assume all classes were in a jar *entertainment.jar*
  - Located in the Lab03.3 directory
  - Run `TelevisionTest` as follows:

```
java -classpath
  C:\StudentWork\workspace\Lab03.3\enertainment.jar
  com.entertainment.TelevisionTest
```

  - The classpath contains the jar file itself
  - The JVM will find the classes inside the jar

```
      Lab03.3
         |
entertainment.jar
```

Lab 7.1: Packages

In this lab, we will practice using packages

Review Questions

1. What is a package? Why do we use packages?
2. What is the relationship between package names and package directory structures?
3. How do you put a class in a package?
4. True or false: using classes that are in a package requires the use of `import` statements.
5. Explain "package private" or "default" access. What keyword is used to signify this access protection level?
6. What is classpath used for?

Lesson Summary

- A **package** organizes and separate classes into **namespaces**
  - Classes are stored in directories mirroring the package name
- Packages are created and used via the following:
  - **`package`** statement: Declares a class is in a package
  - **`import`** statement: Identifies types from another package
    - Import is not required - can always use the fully qualified name
    - Import is also not necessary for classes in the same package
- Packages interact with Java access and encapsulation
  - **Access** is indicated with `public`, `private`, and `protected`
  - The **default access** (package private) provides access to types in the same packages (as well as subtypes)
- The **classpath** defines where Java tools look for types
  - It can contain directories, zip files, and JAR files

Session 8: Composition and Inheritance

Composition

Inheritance

Overriding and Polymorphism

class `Object`

Abstract Classes

Session Objectives

- Understand and use composition
- Understand the characteristics and uses of inheritance and the IS-A relationship
- Explain polymorphism and how it relates to overriding
- Understand the significance of class `Object`
- Understand the characteristics and uses of abstract classes

Composition

Composition

Inheritance

Overriding and Polymorphism

class `Object`

Abstract Classes

Dealing With Complexity and Composition

- We've looked at fairly simple types
  - But types get fairly complex
  - e.g. modeling a car, with all its complex functionality
- One solution - **divide and conquer**
  - Spread the behavior among **smaller types**
  - e.g. use an engine and transmission type in your car
- **Composition** is the assembling (composing) of objects
  - To get our more complex functionality
  - e.g. creating a TV from a `Screen`, `Speaker`, and `Tuner`
- The relationship between the containing and contained part is often called **HAS-A**

Composition

- ***Composition*** or **HAS-A**: Composing an object from other objects, called *subparts* or *components*

```
public class Engine { /* ... */ }
```

```
public class Transmission { /* ... */ }
```

```
public class Car
{
  // a Car object is composed of other objects
  private Engine       engine = null;
  private Transmission tranny = null;

  // construct a Car with Engine and Transmission
  public Car(Engine eng, Transmission trans)
  {
    engine = eng;
    tranny = trans;
  }
}
```

Delegation

- ***Delegation***: Fulfilling a responsibility by using another object
  - Works hand in hand with composition

```
public class Engine {
  public void start() { /* ... */ }
  public void rev() { /* ... */ }
}
```

```
public class Transmission {
  public void shiftTo(int gear) { /* ... */ }
  public void engage() { /* ... */ }
}
```

```
public class Car {
  public void moveTo(String destination) {
    // delegate work to Engine and Transmission
    engine.start();      // engine on previous slide
    engine.rev();
    tranny.shiftTo(1);  // tranny on previous slide
    tranny.engage();
  }
}
```

Benefits of Composition

- Objects are more likely to be correct
  - Smaller, focused on **smaller tasks** and **less complex** to create
- Can change behavior at run-time (**Run-time pluggability**)
  - Substitute a part with a different one
  - e.g., put a new turbo-charged engine in the car
  - Or add new aspects (types) dynamically
  - Interfaces help you remain uncoupled to a specific implementation
    - Covered soon
- **Fosters reuse**: Smaller components are often easier to reuse
  - They have a smaller set of behaviors
  - If programming to interfaces, any type implementing the interface can be used

Issues with Composition

- **More objects** are necessary
  - Several sub-objects instead of one larger object
  - The engine, transmission, the car

- Can be **harder to understand**
  - You may need to look at several sub-parts to understand behavior
  - Run-time pluggability makes it harder to know exactly what is being used
  - Static design is easier to understand

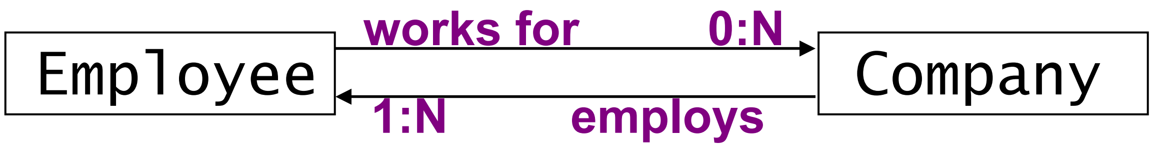
- Potential **run-time inefficiencies**
  - There's another layer between the user of the composite and the underlying object doing the actions
  - Only significant if creating large numbers of objects

Relationships

- **Relationships** are logical connections between objects or types
  - Composition is just one example
  - – There may be many different relationships
  - – Inheritance, or user-defined relationships
- ***Inheritance***: The **IS-A** relationship
  - – A relationship among **types**
  - – Also called the *generalization-specialization* relationship
  - – Covered next
- Composition and inheritance are built into the OO model

Other Kinds of Relationships

- There can be domain-specific relationships
- Consider two classes, `Employee` and `Company`:



- An `Employee` **works for** zero or more companies
  - Denoted as **0:N** in our diagram
- A company **employs** one or more employees
  - Denoted as **1:N** in our diagram
- **works for** and **employs** are **inverse** relationships

[Optional] Lab 8.1: Composition

In this lab, we will practice using composition

Inheritance

Composition

Inheritance

Overriding and Polymorphism

class `Object`

Abstract Classes

Inheritance and Dealing With Complexity

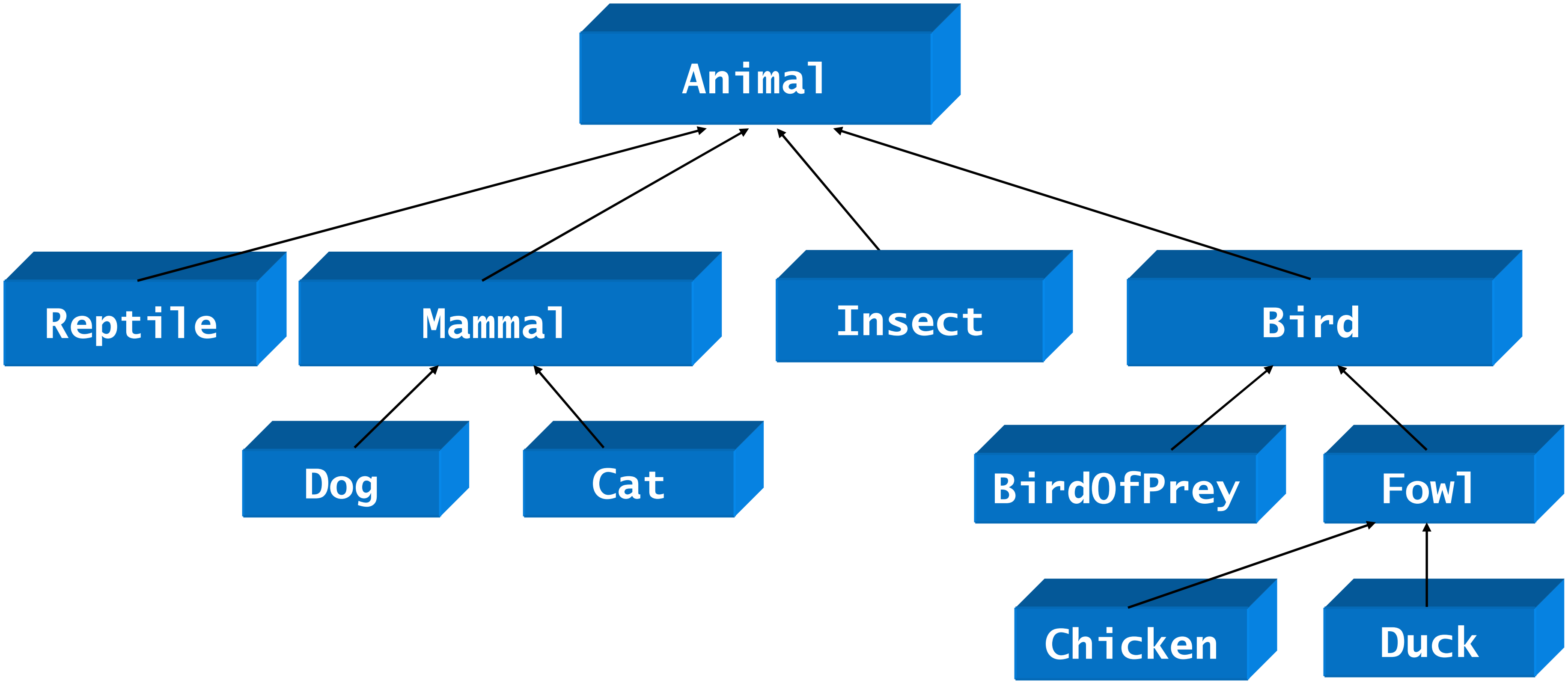
- It's common for different related types to differ in some ways
  - e.g. types keeping time – clock, watch, sundial
  - e.g. different vehicles – car, boat, train

- **Inheritance** is one tool to help with this
  - **Derive more specific types** from other, more general, types
  - The derived type generally specializes or adds behavior
  - A key tool in OO programming

- Inheritance **gathers common features** of related types into a **single general type**
  - Subtypes are defined by their **differences** from the general type

Inheritance Hierarchy

- Inheritance uses levels of abstraction to foster reuse
  - It groups types into **inheritance hierarchies**



The extends Keyword

- Use **extends** to indicate inheritance:
  class *ClassName* **extends** *ClassName*

```
import java.util.Date;

public class Timepiece {
  private Date currentTime;          // Hey – this is composition

  public Date getCurrentTime() {
    return currentTime;
  }
}
```

```
public class AlarmClock extends Timepiece {
  private int snoozeInterval;
}
```

```
AlarmClock myClock = new AlarmClock();
// AlarmClock IS-A Timepiece, you can use TimePiece methods
System.out.println(myClock.getCurrentTime());
```

Inheriting from the Superclass

- A subclass inherits all non-**`private`** methods of its superclass
  - **`private` methods** are **not** inherited
  - **Constructors** are **not** inherited
- An instance of a subclass **contains all of the superclass data**
  - Including its private fields
  - BUT, the superclass private fields **can't be directly accessed** by the subclass

Inheritance and Superclass Data Members

```
import com.mycompany.time.Timepiece;

public class AlarmClock extends Timepiece {
  private int snoozeInterval;
}
```

- An `AlarmClock` instance contain two fields
  - `currentTime` inherited from `Timepiece`
  - `snoozeInterval` that it declares itself
- It also inherits `Timepiece.displayCurrentTime()`
- Here's what an `AlarmClock` instance might look like in memory

`Timepiece part:`	`currentTime`
`AlarmClock part:`	`snoozeInterval`

A Subclass IS-A Superclass

- We can treat a **subclass** instance like a **superclass** instance
  - It inherits all superclass data members and non-private methods
  - An `AlarmClock` **IS-A** `Timepiece`

- You can assign an `AlarmClock` instance to a `Timepiece` variable
  - Or pass an `AlarmClock` to a method parameter of type `Timepiece`

```
Timepiece t = new AlarmClock();
```

Accessing Superclass Members

- Non-**private** superclass members can be accessed in the subclass by name
  - Since an `AlarmClock` IS-A `Timepiece`
  - `private` members **can't** be accessed from the subclass because of the access rules

```
import com.mycompany.time.Timepiece;
public class AlarmClock extends Timepiece
{
  private int snoozeInterval;

  public String toString() {
    // OK to call Timepiece's getCurrentTime() (it's public)
    Date d = getCurrentTime();

    // ERROR: can't access private superclass member variable
    System.out.println(currentTime);
    // Rest of method not shown
  }
}
```

Constructors and Inheritance

- You must often deal with superclass initialization
  - Pass arguments to a superclass constructor via **super()**
  - Only legal as **first statement** of a **subclass constructor**

```
// packages, imports, etc. …
public class Timepiece {
  private Date currentTime;
  public Timepiece(Date d) { // You need a Date to create a Timepiece
    currentTime = d;
  }
}
```

```
// packages, imports, etc. …
public class AlarmClock extends Timepiece {
  public AlarmClock(Date d) {
    // pass d to superclass constructor
    super(d);

    // other AlarmClock constructor code
  }
}
```

Final Classes

- Classes declared **`final`** may not be extended (subclassed)
  - Several `java.lang` classes are `final`
  - Such as `String` and `System`

```
package com.mycompany.time;
public final class Timepiece {
   ...
}
```

```
import com.mycompany.time.Timepiece;
// ERROR: can't derive from a final class
public class AlarmClock extends Timepiece {
   ...
}
```

Lab 8.2: Inheritance

In this lab, we will practice using inheritance

Overriding and Polymorphism

Composition

Inheritance

Overriding and Polymorphism

class `Object`

Abstract Classes

Changing Behavior with Method Overriding

- Sometimes you want to change behavior in a subclass
- You can declare a method with the **same name and signature** (argument list) as the superclass
  - This **overrides** the superclass method
  - Invoking the method on a subclass instance invokes the subclass method

```
public class Timepiece {
  public void displayCurrentTime() { …  }
}
```

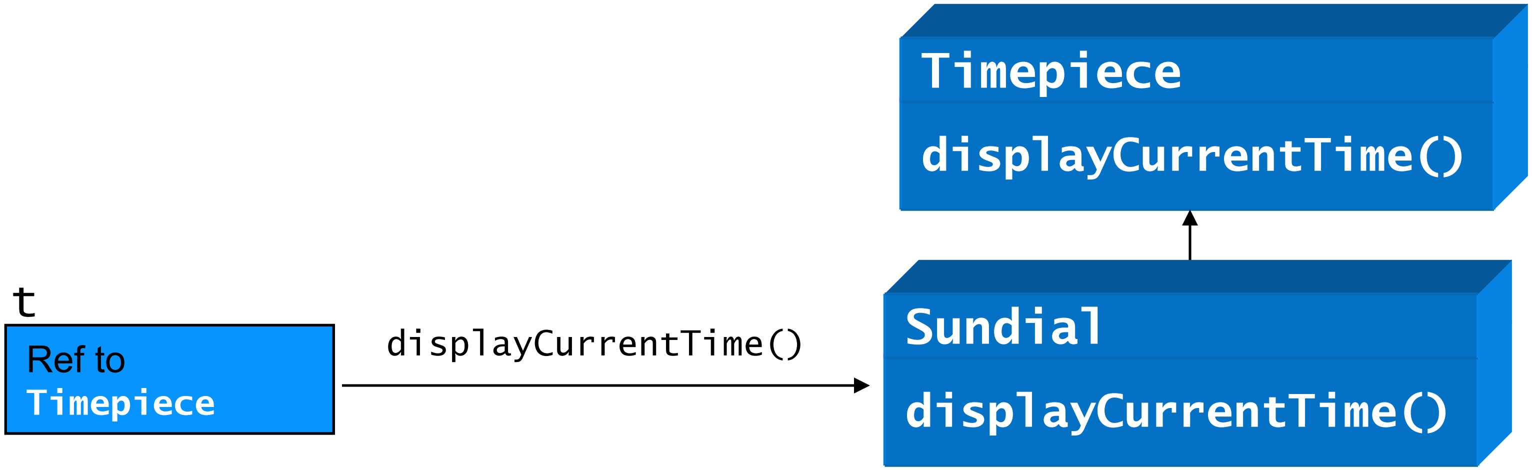
```
public class Sundial extends Timepiece {
  // override Timepiece's displayCurrentTime()
  public void displayCurrentTime() { …  }
}
```

```
Sundial s = new Sundial();
// invokes Sundial's overriding displayCurrentTime
s.displayCurrentTime();
```

OO Concepts - Polymorphism

- **Polymorphism**: Objects of different (related) types can respond to the same method call differently
  - Regardless of the reference type
  - Below, **the `Sundial` method is called**
  - **But, the reference is a `Timepiece` reference**

```
Timepiece t = new Sundial();
t.displayCurrentTime();
```



Polymorphism

- ***Polymorphism*** (***run-time binding***): Dispatches method calls based on the **real** (**run-time**) type of an object
  - **Not** the type of the invoking reference (**compile-time**)
  - The types must be related via inheritance

```
public class Timepiece {
   public void displayCurrentTime() { /* ... */ }
}
```

```
public class Sundial extends Timepiece {
  // override Timepiece's displayCurrentTime()
  public void displayCurrentTime() { /* ... */ }
}
```

```
// A Sundial accessed via a Timepiece reference
Timepiece t = new Sundial();  // a Sundial IS-A Timepiece!
// invoke Sundial's displayCurrentTime() method (Why?)
t.displayCurrentTime();
```

Importance of Polymorphism

- Polymorphism is often used with collections of related objects

```
public class Timepiece { displayCurrentTime()… }
```

```
public class AlarmClock extends Timepiece { displayCurrentTime()…}
```

```
public class Watch extends Timepiece { displayCurrentTime()…}
```

```
public class Sundial extends Timepiece { displayCurrentTime()…}
```

```
// suppose you have an array of Timepieces
Timepiece[] tPieces = new Timepiece[3];
 tPieces[0] = new AlarmClock();
 tPieces[1] = new Watch();
 tPieces[2] = new Sundial();
// use polymorphism to invoke the displayCurrentTime method
// of the different underlying types
for (Timepiece t : tPieces) {  // Iterate with for-each
  t.displayCurrentTime();
}
```

The super Keyword

- The **super** keyword lets code refer directly to their superclass

```
import com.mycompany.time.Timepiece;
public class AlarmClock extends Timepiece {
  private int snoozeInterval

  // overriding method
  public void displayCurrentTime() {
    // OK - calls Timepiece.displayCurrentTime()
    super.displayCurrentTime();
    System.out.println("In the snooze zone: " + snoozeInterval);
  }

  public void useCurrentTime() {

    // ERROR: still can't access private superclass variable
    System.out.println(super.currentTime);
  }

}
```

Access Control - protected Access

- Java access controls work with inheritance

- A **protected** field is **accessible within a subclass**
  - Even if the subclass is in another package
  - It's also accessible in the package containing the defining class
  - Protected methods give subclasses (only) a **chance to customize behavior**
  - Protected fields aren't very useful

- Overriding classes **can't reduce visibility** for a method
  - **No** overriding of a **public method** with a **private** one
  - The compiler will complain
  - You can open access wider (i.e. from private to public)

Access Control - protected Access

```
package com.mycompany.time;
public class Timepiece {
  private Date currentTime;

  public void displayCurrentTime() {
    System.out.println(currentTime);
    // Give subclasses a chance to do some extra display
    displayExtra();  // Polymorphism!  See note
  }

  protected void displayExtra() {  }
}
```

```
import com.mycompany.time.Timepiece;

public class AlarmClock extends Timepiece {
  int snoozeInterval;

  // Subclasses can access/override protected members
  protected void displayExtra() {
    System.out.println("In the snooze zone: " + snoozeInterval);
  }
}
```

@Override

- Good practice to indicate you are overriding a method
  - Done with **@Override** - an annotation (annotations covered later)
  - An `@Override` annotated method MUST override a superclass method
- Protects against errors in the overriding method's signature
  - If you don't override (e.g. make a mistake in the signature), it's a compile time error
- Documents that you're overriding a method (for clarity)

```
public class Timepiece {
   public void displayCurrentTime() { /* ... */ }
}
```

```
public class Sundial extends Timepiece {
  @Override
  public void displayCurrentTime() { /* ... */ }
}
```

Lab 8.3: Polymorphism

In this lab, we will practice using polymorphism

`class Object`

Composition

Inheritance

Overriding and Polymorphism

`class Object`

Abstract Classes

Class Object

- All Java classes ultimately derive from a single ***root class*** – **`Object`**

```
package com.mycompany.time;
// the following is the same as
// 'public class Timepiece extends Object'
public class Timepiece {
   ...
}
```

```
import com.mycompany.time.Timepiece;
// inherits from Object through Timepiece
public class AlarmClock extends Timepiece {
   ...
}
```

Methods of Class `Object`

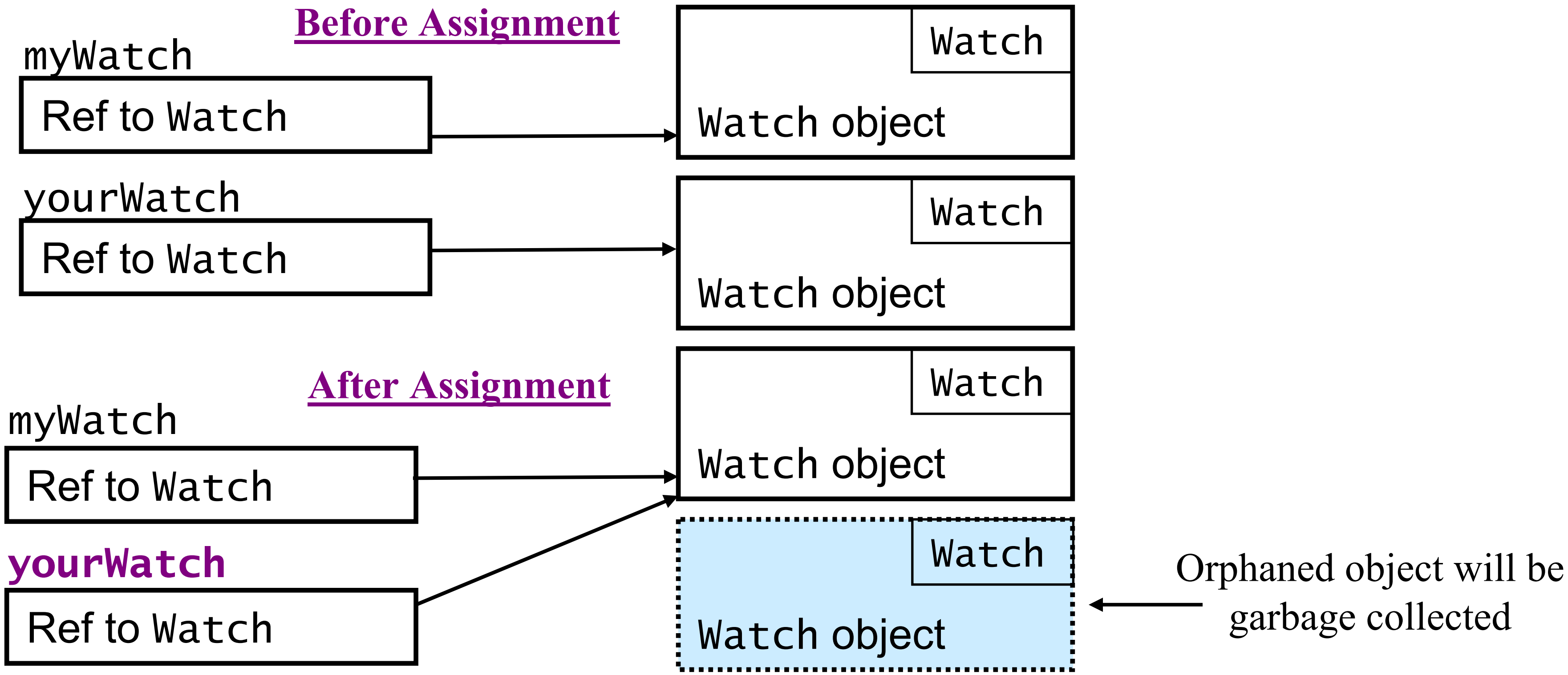
- Since **all** classes inherit from `Object`, its methods are available for **any** instance of **any** class

- Common methods include:
  - `public boolean` **`equals`**`(Object obj)`: Compare two instances for **equality of their contents**
  - `public String` **`toString`**`()`: Return a `String` representing the instance value in some textual form
  - `protected Object` **`clone`**`()`: Return a "shallow" copy of the object
  - `protected void` **`finalize`**`()`: Cleanup code executed when this object is garbage collected (not often used)
  - Some others are `hashCode()`, `getClass()`, `notify()` …

Automatic Storage Management

- Java reclaims object storage with ***garbage collection***
  - By automatically detecting when an object is not referenced

```
Watch myWatch = new Watch();        // object1
Watch yourWatch = new Watch();      // object2
yourWatch = myWatch;                // assignment
```



Abstract Classes

Composition

Inheritance

Overriding and Polymorphism

class `Object`

Abstract Classes

Abstract Classes

- ***Abstract classes*** are classes which can't be instantiated

```
public abstract class Timepiece {
  /* variables, method definitions, etc. */
}
```

- Abstract classes may, and usually do, have abstract methods
  - Abstract methods defined in next slide

```
// Timepiece cannot be instantiated
// a subclass (AlarmClock, etc.) must implement
// the abstract methods
public abstract class Timepiece {
  public abstract void displayCurrentTime();
}
```

- Ensures that all subclasses support the method
  - With implementation deferred to the subclass

Abstract Methods

- ***Abstract methods*** define no implementation
  - They end with a semicolon (**;**)
  - They have no method body
- Subclasses override abstract methods with ***concrete methods***

```
// displayCurrentTime() is abstract - we don't
// know how to implement it until we get to a more
// specific type (AlarmClock, Sundial, etc.)

public abstract void displayCurrentTime();
```

Using Abstract Classes

- Abstract classes often specify a ***protocol*** or interface
  - A set of standard methods
- To instantiate, you must create a ***concrete subclass***
  - It must override all abstract methods in order to be concrete

```
public abstract class Timepiece {
  public abstract void displayCurrentTime();
}
```

```
public class Sundial extends Timepiece {
  public void displayCurrentTime() {
    // implement Sundial's displayCurrentTime() here
  }
}
```

Review Questions

1. What is composition? List some of its advantages and disadvantages.
2. Explain delegation and its role in composition.
3. What is inheritance
4. Which two things are **not** inherited by a subclass?
5. True or false: a `final` class may not be instantiated directly.
6. What is polymorphism, and what are its prerequisites?
7. Why is polymorphism useful?

Lesson Summary

- **Composition assembles objects** from other objects
  - The composing object can **delegate**, or pass on, responsibility for its behavior to its contained objects
  - Result: **Smaller objects**, well defined behavior, and more reuse
- **Inheritance** derives more specific types from general types
  - Shares common features and supports treating subtypes as the same general supertype
  - A subclass does not inherit private members and constructors
  - A final class does not allow inheritance, but can be instantiated
- **Polymorphism** dispatches method calls at run time
  - Based on the real type of an object, not the reference
  - It allows you to abstract away a type hierarchy - treating all objects as a general type, while preserving specialized behavior

Session 9: Interfaces

Interface Basics

Default Methods and static Methods

Session Objectives

- Understand the similarities between interface types and class types
- Use interface types the same way that class types are used
- Explain the role that interfaces play in "programming by contract"
- Define and implement an interface

Interface Basics

Interface Basics

Default Methods and static Methods

What if All You Have to Share is an Idea

- You often know what a type will do
  - You know **what** its behavior (methods) are
  - But you don't know **how** it will do it
- Or many related types will implement behavior **differently**
  - And you want to treat all those types the same
- For example, a `Timepiece` displays time
  - But different timepieces display it differently, and have no shared implementation
  - A clock, a sundial
  - What about a cell phone?
  - You can make `Timepiece` an **interface**



Interface Types

- Interfaces let you to specify a type **separately from any implementation**
  - It is an abstract type that can specify the kind of behavior
    - But not its implementation
  - Interfaces often define ***roles*** played by objects
  - A class can define **how** a type fulfills the role
    - Via fields and method implementations

- An ***interface*** defines a type similar to a class
  - But its **methods are abstract**
  - They can't be instantiated directly with `new`
  - It can have properties, but they're all **`static final`** **constants**

Interface Definitions

- Defined via the **interface** keyword
  - Similar to the `class` keyword
  - We'll illustrate this using types for a shipping company

```
// definition of interface Moveable
package com.mycompany.shipping;
public interface Moveable {
  // ...
}
```

- Interface methods have no body
  - They're implicitly abstract, with no implementation

```
package com.mycompany.shipping;
public interface Moveable {
  public void moveTo(String dest);
}
```

The `implements` Keyword

- Classes can **`implement`** an interface in their definitions
  - As shown below
  - You provide implementations for every method of the interface
  - If you don't, the class must be declared abstract

```
import com.mycompany.shipping.Moveable;
public class PosterTube implements Moveable
{
  // provides an implemented moveTo(String) method
  public void moveTo(String dest)
  {
    // PosterTubes's implementation
  }
}
```

Example of Using Interface Types

```
class MovingCompany {

  // Moveable is an interface type
  Moveable[] goods = null;

  MovingCompany(Moveable[] goodsIn) {
    goods = goodsIn;
  }

  void deliverAllGoods(String location) {
    for (Moveable m : goods) { // Iterate with for-each
      m.moveTo(location);
    }
  }
}
```

Interface Types as References

- Remember - an interface type is similar to a class type
  - An instance may be "viewed" by other objects by the **interface types** that it implements
  - And **not** by its actual class type
  - For example, a moving company might not care about what **exactly** it is moving, just that the items are `Moveable`.
  - `Moveable` is a type, but it is an **interface type**, not class type

Interface Types as References

- Interface types can be used as **reference variable types**, but cannot be instantiated or used as object types

```
Moveable m = new PosterTube();
```

- Interface types can be used as **parameters to methods**

```
public void moveObject(Moveable m) { ... }
```

- Interface types can be used as **return types** from methods

```
public Moveable getMovedObject()
{
   return m;
}
```

Extending Interfaces

- **Subinterfaces** extend other interfaces, via the **extends** keyword
  - The **IS-A** relationship with interfaces
  - You can extend multiple interfaces (unlike class inheritance)
  - A class implementing a subinterface must implement all the interfaces that the subinterface extends (**IS-A**)

```
// A generic movable
public interface Moveable {
  public void moveTo(String dest);
}
```

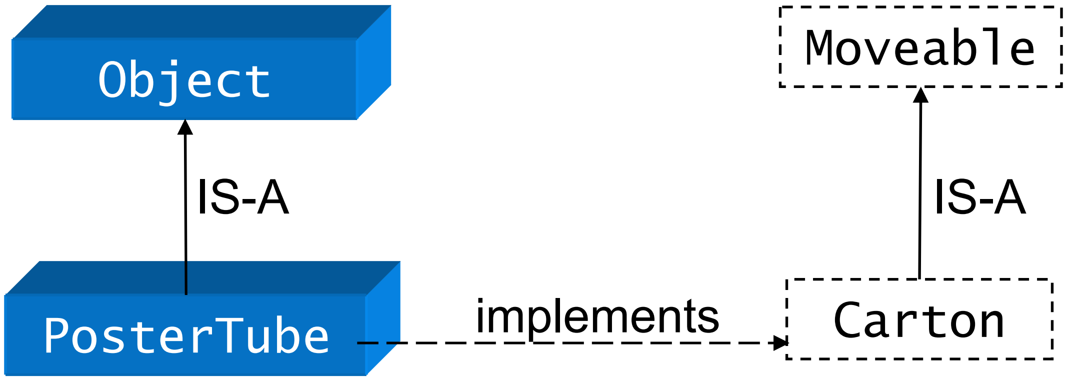
```
// A moveable that will go onto a truck
public interface Carton extends Moveable {
  public float getSize();
}
```

Implementing Extended Interfaces

```
// PosterTube implements Carton and thus Moveable, also
public class PosterTube implements Carton
{
  // from interface Moveable
  public void moveTo(String dest) {
    // ...
  }

  // from interface Carton
  public float getSize() {
    // ...
  }
}
```

Example of Using Interface Types

```
interface Moveable { /* ... */ }
interface Carton   extends Moveable    { /* ... */ }

class Car          implements Moveable { /* ... */ }
class PosterTube   implements Carton   { /* ... */ }
class ShippingBox  implements Carton   { /* ... */ }
class WardrobeBox  extends ShippingBox { /* ... */ }
```



Example of Using Interface Types

```
class MovingCompany {

  // Carton and Moveable are interface types
  Carton[] cartons = null;
  Moveable[] goods = null;

  MovingCompany(Carton[] cartonsIn, Moveable[] goodsIn) {
    cartons = cartonsIn;
    goods = goodsIn;
  }

  void deliverAllGoods(String location) {
    float totalSize = 0.0F;
    for (int i = 0; i < cartons.length; i++) {
      totalSize += cartons[i].getSize();
      cartons[i].moveTo(location);
    }
    for (Moveable m : goods) {
       m.moveTo(location);
    }
  }
}
```

Example of Using Interface Types

```
class GetMoving
{
  public static void main(String[] args)
  {
    Carton[] boxes = { new PosterTube(), new ShippingBox(),
                       new WardrobeBox() };

    Moveable[] bigStuff = { new Car() };

    MovingCompany acme = new MovingCompany(boxes, bigStuff);

    acme.deliverAllGoods("San Francisco");
  }
}
```

Interfaces are Abstract

- Interfaces are implicitly `abstract`
  - You can declare this explicitly, though you generally don't't
  - The definition below is equivalent to one without abstract

```
// 'abstract' legal, but generally not included
public abstract interface Moveable {
  // ...
}
```

  - `abstract` also legal on methods but generally not used
  - The definition below is equivalent to one without abstract

```
public interface Moveable {
  // 'abstract' legal, but generally not included
  public abstract void moveTo(String dest);
}
```

Data Members in Interfaces

- Interfaces can't declare instance data
  – If you need instance data, use a class
- Interface data members are **implicitly static** and **final**,
  – They're usually declared as such, for clarity
  – It's legal to leave out `final` and `static` - the compiler will just add them in

```
package com.mycompany.shipping;
public interface Moveable
{
  // static and final are usually included
  // note the ALL_CAPS convention for class constants

  public static final String HOME_LOCATION = "HQ Office";
  public void moveTo(String dest);
}
```

Implementing Multiple Interfaces

- A class can implement multiple interfaces

```
public class Car implements Moveable, Serviceable
{
  // from interface Moveable
  public void moveTo(String dest) {
    // ...
  }
  // from interface Serviceable
  public void serviceEngine() {
    // every 30,000 miles turn on "Check Engine" light
  }
}
```

- Has many of the advantages of multiple inheritance, which is not supported in Java
  - Less complex because the interfaces declare a role, not an implementation

Lab 9.1: Interfaces

In this lab, we will work with interfaces - both creating and using them

Default Methods and static Methods

Interface Basics

Default Methods and static Methods

Default Methods in Interfaces (Java 8+)

- **Default methods** provide default implementations
  - Defined in the interface declaration
  - The method is **not** abstract
  - Extending classes **don't need to implement** a default method
    - They inherit it, but may override it with their own definition
- Below, we add `getCurrentLocation()` to `Moveable`
  - The **`default`** keyword and the implementation indicate it's a default method (1)

```
import com.mycompany.gps.Location;  // Details not shown

public interface Moveable {
  public void moveTo(String dest);
  default public Location getCurrentLocation() {
    return new Location (/* … */); // Details not shown
  }
}
```

Using Default Methods

- Below, `Car` implements `Moveable`
  - And inherits the default `getCurrentLocation()`
  - At bottom, we create a car, and call `getCurrentLocation()` on it
  - The default implementation is used, and everything works

```
public class Car implements Moveable {
  public void moveTo(String dest) { /* … */ }
}
```

```
class GetMoving {
  public static void main(String[] args) {
    Car theCar = new Car();
    System.out.println(theCar.getCurrentLocation());
  }
}
```

Uses of Default Methods

- **Provide a common implementation** for reuse
  - Classes easily inherit and use the functionality
  - e.g., `java.lang.Iterable` defines a default `forEach()` (1)

- Allow for **easy evolution** of interfaces
  - As we added `getCurrentLocation()` to `Moveable`
  - Adding a regular interface method requires defining it in all `Moveable` implementations and recompiling them (2)
  - When adding default methods, **existing implementations continue working unchanged**
    - **Binary compatibility** is maintained
    - As with our `Car` example

Static Methods in Interfaces (Java 8+)

- **Static methods** are legal in interfaces
  - Static methods in an interface work the same as in a class

- For example, the `Comparator` interface defines the static **`naturalOrder()`** method, as shown below
  - `naturalOrder()` is a **factory method** generating an object
  - Factory methods are often static, and can now be defined in an interface
  - The details of `Comparator` are not relevant (1)

```
package java.util;

// Simplified, and with many details omitted …
public interface Comparator<T> {
  static Comparator<T> naturalOrder() { /* … */ }
}
```

[Optional] Lab 9.2: Default Method

In this lab, we will add a default method to an interface and see how it works

Review Questions

1. How does "programming by contract" apply to interfaces?
2. What keyword is used for a class to "sign an interface contract?"
3. True or false: interfaces can be placed in packages.
4. True or false: interfaces can only have a default or no-argument constructor.
5. True or false: interfaces can exhibit inheritance characteristics similar to classes.

Lesson Summary

- **Interface** types define an abstract set of methods
  - A "contract" that an implementing class must fulfill
  - An implementing class `extends` an interface to sign the contract
- They are similar to abstract classes, and can use many class capabilities
  - e.g., they can be put in a package and use inheritance
  - They can have default method implementations and static methods (Java 8+)
- There are differences from abstract classes
  - No constructors
  - All methods of a public interface are implicitly public

Session 10: Exceptions

Exception Hierarchy

Handling Exceptions - `try` and `catch`

Session Objectives

- Understand what exceptions are used for and how they are used
- Differentiate between checked and unchecked exceptions
- Learn how to throw exceptions
- Learn how handle exceptions with `try-catch-finally`
- Define your own exception classes

Exception Hierarchy

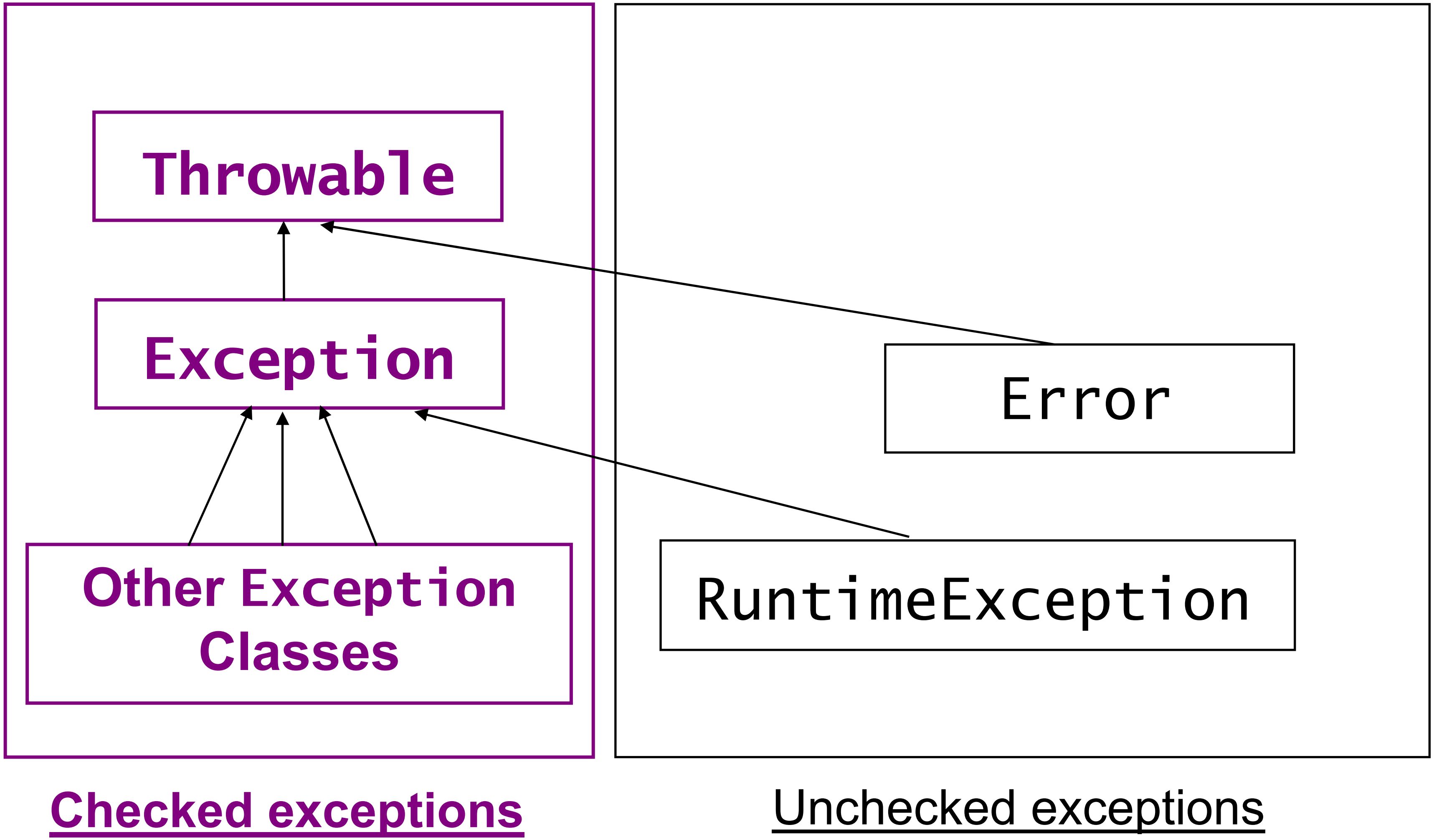
Exception Hierarchy

Handling Exceptions - `try` and `catch`

Overview of Exceptions

- Exceptions **notify calling code of some unusual condition**
  - They generally signal errors - in your code, user input, a database, etc.
- **Exceptions are objects**
  - They're not errors - but used to handle errors
  - The exception type indicates the kind of exception
  - The exception object may contain information about the exception
- **Exceptions cause a jump in program flow**
  - The program stops and responds to the exception

Exception Hierarchy



Checked and Unchecked Exceptions

- **Checked exceptions**: Must be handled or declared in a `throws` clause
  - "Checked" means checked by the compiler
  - These **must be handled in some way**
    - The compiler enforces this
    - If you don't handle them, your code **won't compile**

- **Unchecked exceptions**: All other exceptions
  - You don't need to handle unchecked exceptions
    - Meaning that the compiler won't check on them
  - You may decide to handle them anyway

Core Exception Classes

- `class` **`Throwable`**: Superclass of all errors/exceptions
  - Two main subclasses, `Exception` and `Error`
- **`Exception`**: Programs **must** handle these (**checked**)
  - Defines an unchecked branch - `RuntimeException`
- **`Error`**: Abnormal events that should **not normally occur**
  - e.g. - out-of-memory condition
  - You generally don't handle `Errors` in application code
- **`RuntimeException`**: `Exception` subclass which cannot be foreseen before runtime
  - It (and its subclasses) are **unchecked** exceptions
  - e.g. `NullPointerException`

Handling Exceptions
`try` and `catch`

Exception Hierarchy

Handling Exceptions - `try` and `catch`

Handling Exceptions with `try` and `catch`

- To handle exceptions you:
  - Designate blocks of protected code with **`try`**
  - Create exception "handlers" with **`catch`**
  - **Exception handler**: Code executed when the exception occurs
    - It "handles" the situation in some manner

- General form for handling exceptions:

`try {` *Block* **`}`**
`catch (` *Argument* **`) {`** *Block* **`}`**

  - The argument to `catch` must be a `Throwable`
    - Or a subclass of `Throwable`
    - It's usually a subclass of `Exception`

Exceptions and Program Flow

- Exceptions cause a jump in program flow
  - Below, the `FileReader` constructor & methods can throw `FileNotFoundException` and `IOException`

```
// FileReader and Exceptions are in package java.io
// code fragment
try {             // If foo.txt does not exist -> EXCEPTION
  FileReader fr = new FileReader("foo.txt");  // 1
  // Do something with fr
  fr.close();  // you need to close streams
}                                           // control passes here
catch (FileNotFoundException e) {              // 2
  System.out.println(e);
}                                              // 3
catch (IOException e) {
  System.out.println(e);
}                                           // then to here
// statement that follows catch block(s)       // 4
```

Variable Scope

- The Exception handler introduces a new variable scope
  - Within it, you can access the exception object

Class scope

Method parameter and local variable scope

Exception handler parameter scope

```
class ScopeSlide {
    int x = 56;
    public void scopeMethod(int num) {
        try {
            int y = x + num;
            System.out.println("How many people?");
            System.out.println("This many: " + y);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The throws Clause

- A method may handle an exception with `try-catch` (seen earlier) or pass it on to the caller
- A **throws** clause **must** declare any **checked exceptions** a method may throw
  - Generally resulting from code in the method body
  - As shown below

```
// code fragment
public void workWithFoo()
   throws FileNotFoundException, IOException {
    // FileReader() and close() may throw exceptions
    // Not caught, so they must be in throws clause above
    FileReader fr = new FileReader("foo.txt");
    // Do something with fr …
    fr.close();  // you need to close streams
}
```

Throwing Exceptions with throw

- Throw an exception via the **throw** keyword
  - Signals an exception in the code
  - `throw` takes one argument, which must be an `Exception`

throw *Exception*;

  - You need an instance of an Exception (often instantiated in your code)
  - You generally provide a message in the constructor, **indicating the reason it's thrown**

```
public void setSnoozeInterval(int snoozeIn)
    throws Exception {
  if (snoozeIn >=0 && snoozeIn <= 100) {
    // set snooze interval as appropriate
  }
  else {
    throw new Exception("snoozeInteval " + snoozeIn
                        + " is invalid");
  }
}
```

User-Defined Exceptions

- Define your own exception classes for app-specific exceptional conditions
  - By subclassing **Exception**

```
public class InvalidSnoozeIntervalException extends Exception {
  InvalidSnoozeIntervalException(){ }
  InvalidSnoozeIntervalException(String message) {
    super(message);
  }
}
```

```
public class AlarmClock {  // Much detail omitted …
  public void setSnoozeInterval(int snoozeIn)
               throws InvalidSnoozeIntervalException {
    if (snoozeIn >=0 && snoozeIn <= 100) {
       // set snooze interval as appropriate
    } else {
      throw new InvalidSnoozeIntervalException("snoozeInteval " +
                snoozeIn  + " is invalid");
    }
  }
}
```

User-Defined Exceptions

- User-defined exceptions work exactly the same
  - Below, we call `setSnoozeInterval()`, and catch the `InvalidSnoozeIntervalException` that may be thrown
  - We also extract the exception's message via `getMessage()`

```
public void wakeUp(AlarmClock c) {
  try {
    c.setSnoozeInterval(10);
  }
  catch (InvalidSnoozeIntervalException e) {
      String msg = e.getMessage();
      System.out.println(msg);
  }
}
```

Multiple catch Blocks

- A `try` block can have multiple `catch` blocks (handlers)
- If an exception is thrown in the `try`, the following occurs:
  - The `catch` blocks' arguments are examined (in the order the catch blocks are written)
  - The first `catch` block argument supporting a legal assignment of exception is chosen to handle it
  - **Be careful when ordering your `catch` blocks**
    - **Put the most specific exceptions first**
    - Otherwise, succeeding exceptions are never reached

Multiple catch Blocks

```
try {
  AlarmClock c = new AlarmClock();
  c.setSnoozeInterval(10);       // 1
}

// catch specific exception
catch (InvalidSnoozeIntervalException se) {   // 2
  ...
}

// catch generic Exception
catch (Exception e) {
  // any other Exception, like oversleeping
}
System.out.println("We're awake");       // 3
```

- What happens if `InvalidSnoozeIntervalException` is thrown at 1?
  - What happens if a different exception is thrown at 1?

`finally` Block

- `try` blocks can also have an associated **`finally`** block
  - The code in a `finally` is executed **after** any code in a `try` or `catch`
- `finally` **ensures** that a block of code is executed
  - Executed **no matter how control leaves** the `try` and `catch` blocks
- What happens if an `InvalidSnoozeIntervalException` is thrown at **1**?

```
void wakeUp(AlarmClock c) throws InvalidSnoozeIntervalException {
  try {
    c.setSnoozeInterval(200);                        // 1
  }
  finally {                                          // 2
    System.out.println("Guaranteed to execute");
    System.out.println("Wake UP NOW !!!");
  }
}
```

Runtime Exceptions

- Runtime exceptions do not have to be handled with try-catch
  - Nor do they need a throws clause
  - `IllegalArgumentException` is defined in `java.lang`
  - It is a subclass of `RuntimeException`

```
public class AlarmClock {  // Much detail omitted …
  public void setSnoozeInterval(int snoozeIn) {
    if (snoozeIn >=0 && snoozeIn <= 100) {
       // set snooze interval as appropriate
    } else {
      throw new IllegalArgumentException("snoozeInteval " +
                snoozeIn  + " is invalid");
    }
  }
}
```

```
public void wakeUp(AlarmClock c) {
    c.setSnoozeInterval(10);
}
```

Multicatch (Java 7+)

- You can catch multiple exceptions in one catch
  – Separate them with an **|**
  – Useful when doing the same thing for each exception
  – For example, just logging it
- We show this below for two exceptions
  – The code that throws the exception is just normal code

```
try {
  // Code that can throw two exceptions IOException
  // or InvalidSnoozeIntervalException
  // Detail omitted
}
// catch both exceptions in one catch
catch (InvalidSnoozeIntervalException | IOException e) {
  e.printStackTrace();
}
```

Using try-with-resources (Java 7+)

- `try-with-resources` closes resources like a `FileReader/Writer`
  - Any resource declared in the try **is automatically closed** when the try ends
  - We show this below for a `FileReader`

```
// code fragment
try ( FileReader fr = new FileReader("foo.txt"); ) {
  // Use fr somehow
  // No fr.close() needed
}
catch (FileNotFoundException e) {
  System.out.println(e);
}
catch (IOException e) {
  System.out.println(e);
}
```

Lab 10.1: Using Exceptions

In this lab, we will work with exceptions - both throwing and catching them

Review Questions

1. What is the superclass of all exceptions?
2. Explain the difference between checked and unchecked exceptions
3. True or false: exceptions are objects, but they are not instantiated with `new`.
4. True or false: the order in which you place your `catch` blocks has no significance.
5. True or false: `finally` blocks are used when you want to implement an optional code segment.
6. Explain the difference between `throw` and `throws`.

Lesson Summary

- **Exceptions** are objects that form an inheritance hierarchy
  - `Throwable` is the root class for all errors and exceptions
  - `Exception` extends `Throwable` and is the root of the exception hierarchy
  - Exception objects are created using new, and "thrown" using a `throw` statement
- Exceptions deriving from `RuntimeException` are **unchecked** exceptions
  - They don't need to include exception handling when thrown
- **All other exceptions** are **checked** exceptions
  - They need to include exception handling
  - Either a `try/catch` block, or a `throws` clause in the method signature

Lesson Summary

- Handle exceptions with `try/catch` blocks
  - With multiple handlers, put the most specific exceptions first

- Use a `finally` block for code that is **ALWAYS** executed
  - No matter how the try block finishes

Session 11: Collections and Generics

Overview

`List` and `ArrayList`

Autoboxing and Collections of Object

Other Collection Types

`Iterator`

More About Generics

The Collections Class

Session Objectives

- Discuss the Java Collections Framework
- Be familiar with the key collection interfaces, `Collection`, `Set`, `List`, and `Map` and the differences between them
- Use common collection implementations such as `ArrayList` and `HashMap`
- Learn how for-each and autoboxing make using collections easier
- Learn the principles of generic classes and how they're used in collections

Overview

Overview

`List` and `ArrayList`

Autoboxing and Collections of Object

Other Collection Types

`Iterator`

More About Generics

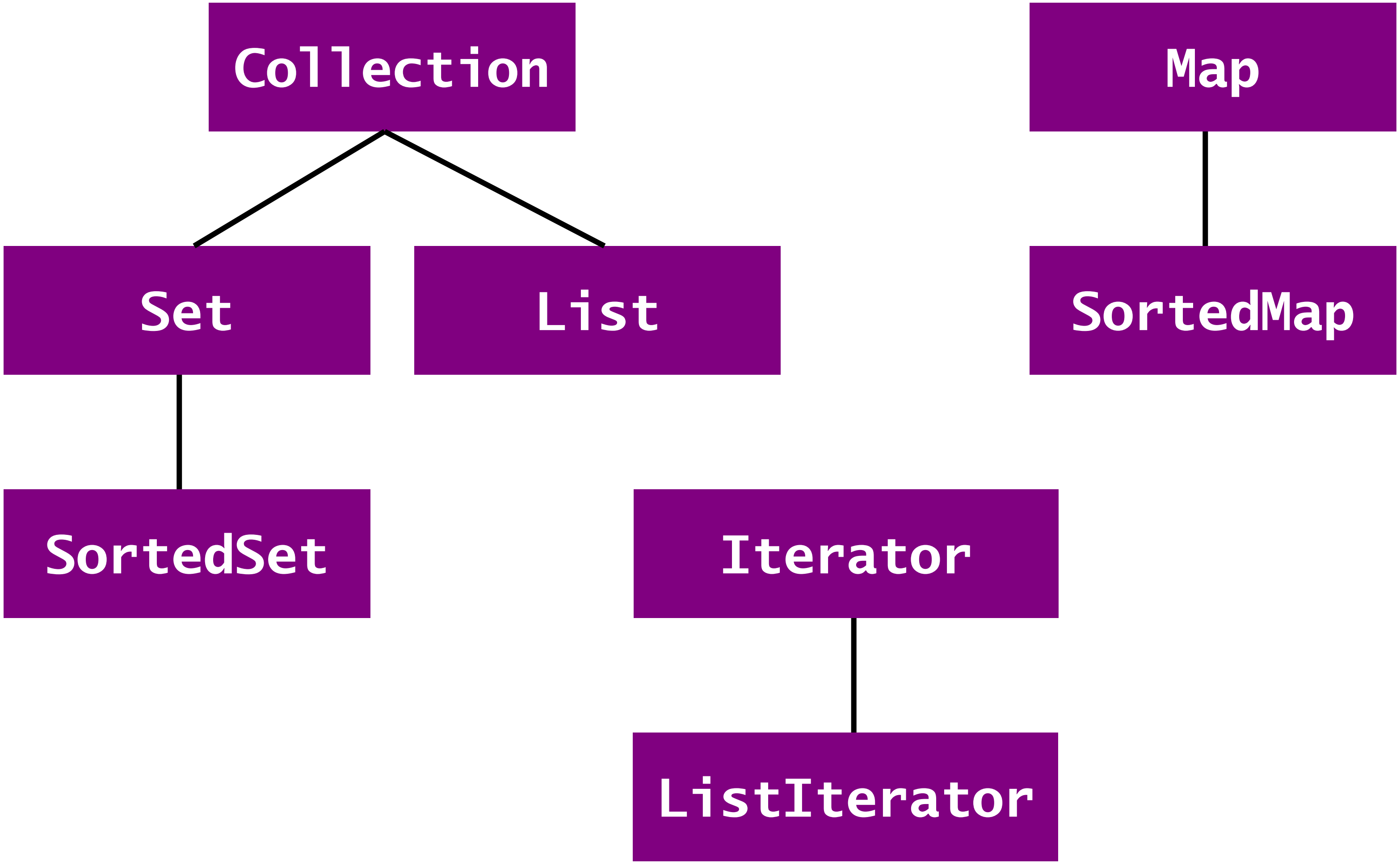
The Collections Class

Java Collections Framework Overview

- An architecture for representing and manipulating collections
  - A ***collection*** represents a group of objects, (its elements)
  - The collection classes are in the package **`java.util`**
  - They supplement array's capabilities

- Arrays and collections are fairly different:
  - Array processing is very fast
  - Array size is declared and cannot change
    - Collections can grow
  - Arrays can store primitives
    - Primitives must be "wrapped" to be added to a collection
    - **NOTE: Java can autobox primitive types**
  - Array has a length attribute – collections have capacity and size

java.util Core Collection Interfaces



Collection Interface

- Represents a grouping of objects (***elements***)
  - Root of the collection types (most general type)
  - Sub-interfaces and classes have different characteristics
    - e.g. no duplicates (`Set`) or ordering (`List`)

- No direct `Collection` implementations
  - Sub-interfaces, such as `Set` and `List`, have implementations

- Common methods include:
  - `boolean` **`add`**`(E o)` : Add o to collection *
  - `boolean` **`contains`**`(Object o)`: Return true if o in collection
  - `int` **`size`**`()`: Return number of elements in collection
  - `boolean` **`remove`**`(Object o)`: Remove o from collection
  - See the javadocs for all the (many) methods in `Collection`

Generics and Type-Safe Collections

- Java ***generics*** provide type safety for collections
  - Notation: **`CollectionClass<Type>`**
    read as **"CollectionClass of Type"**
- Below, we illustrate **`ArrayList<String>`**
  - Read as "`ArrayList` of `String`"
  - Note that `<String>` is part of the type declaration
    - In the new expression, constructor args go after `<String>`
  - We'll cover `ArrayList` itself later
- Always use type-safe collections - they are safer and easier

```
// Generics: Compiler only allows Strings to be added to brands
ArrayList<String> brands = new ArrayList<String>();
brands.add("RCA");                 // OK
brands.add(new Television());  // ERROR - will not compile
```

List and ArrayList

Overview

List and ArrayList

Autoboxing and Collections of Object

Other Collection Types

Iterator

More About Generics

The Collections Class

List Interface

- **List**: An **ordered collection** (or ***sequence***)
  - Allows access by index (get, add, remove)
  - Zero-based indexing, like Java arrays
  - Duplicates allowed
  - Some methods have additional stipulations to maintain order

- Common methods include:
  - `boolean add(E el)`: Add el to end of list
  - `void add(int index, E el)`: Add el at specified position
  - `E get(int index)`: Return element at index
  - `int indexOf(Object o)`: Return index of o, -1 if not found
  - `E remove(int index)`: Remove element at index
  - And, of course, all the collection methods

ArrayList

- **ArrayList** implements List
  - Based internally on an array
- **Resizable**, and can change size dynamically
  - The initial **capacity** can be set with its constructor
- Below, we create an ArrayList with a capacity of 2
  - We add in two Television instances to it

```
import java.util.ArrayList;  // Much detail omitted …

Television tv1 = new Television("RCA", 10);
Television tv2 = new Television("Hitachi", 10);
// Create ArrayList with capacity of two, & add Televisions
ArrayList<Television> tvs = new ArrayList<Television>(2);
tvs.add(tv1);
tvs.add(tv2);
```

Using ArrayList - Example

- Below, we add two elements to an `ArrayList<Television>`
  - We use `get()` to retrieve a television at a specific index
  - **No casting** is needed with generics

```
import java.util.ArrayList;
class Televisions {
  public static void main(String[] args) {
    ArrayList<Television> tvs = new ArrayList<Television>();
    tvs.add(new Television("RCA", 10));
    tvs.add(new Television("Hitachi", 5));
    System.out.println("current size is " + tvs.size());

    // access the second element of the list
    Television tv = tvs.get(1);
    System.out.println("Second Television is " + tv);
  }
}
```

for-each

- The **for-each** loop works for collections

```
ArrayList<Television tvs = new ArrayList<Television>();
// Assume you add tvs

// for-each Television curTV in tvs
for (Television curTV : tvs) {   // Simple and easy iteration
  System.out.println(curTV);
  curTV.setVolume(0);
}
```

- Read the loop as **"for each Television curTV in tvs"**
  - It goes through the collection, and initializes `cur` with the current television on each iteration
  - It's the preferred, and easy, way to iterate through collections

Autoboxing and Collections of Object

Autoboxing

- Collection classes contain `Objects`
  - You can't have a collection of `ints`, but **can** have a collection of `Integer` objects
  - So "box" the `ints` into `Integers` and put them in the collection
  - Not difficult, but clumsy
- ***autoboxing/unboxing*** eliminates manual conversion between primitive and wrapper types
  - The compiler does the conversion, as shown below

```
// Manually wrapping/unwrapping
Integer age = new Integer(38);  // int -> Integer
int age_int = age.intValue();   // Integer -> int
```

```
// Autoboxing/unboxing
Integer age = 38;   // Compiler generates Integer instance
int age_int = age;  // Compiler generates intValue call
```

Using Autoboxing/Unboxing - Example

- Autoboxing works with collections also
  – Declare the collection and put primitives in directly!

```
// Autoboxing:
List<Integer> ages =
   new ArrayList<Integer>();    // generic/type-safe collection
ages.add(38);                   // autoboxing: no conversion
ages.add(17);                   // autoboxing: no conversion

for (int age : ages) {          // Note - age is of type int
  int time_to_retire = 65-age;  // unboxing: no conversion
}
```

```
// Without autoboxing - just for comparison
List ages = new ArrayList();
ages.add(new Integer(38));
// ...
for (Object cur : ages ) {  // Iterate using for-each
  int age = ((Integer) cur).intValue();  // messy
  int time_to_retire = 65-age;
}
```

Summarizing Collection Features

- These language features are independent of each other
- You often use multiple features together, but don't have to

Feature	Provides	Removes
generics	type-safe collections	casting
autoboxing	(illusion of) collections of primitives	primitive/wrapper conversions
for-each	compact, easy looping	iterator code

Collections of Object

- You can use collections without type parameters

```
ArrayList brands = new ArrayList();
```

  – This is a collection of `Object`
  – **NOT recommended** - available for backwards compatibility
  – You may still see code like the above [(1)]

- A collection of `Object` is **less safe** than a type-specific one
  – **add** basically looks like this:

```
add(Object o)
```

  – All types are ultimately `Objects`, so the compiler can't enforce type safety
  – e.g. you can add a television to a collection you think has strings

Issues with Collection of Object

```
ArrayList brands = new ArrayList(); // intend to hold Strings
brands.add("RCA");
brands.add(new Television());        // not a String
...
// elsewhere in the application, we process the collection
for (Object cur : brands) {
  String brand = (String) cur;    // vulnerable
}
```

- This code fails with a runtime exception
  - `ClassCastException` is thrown at `(String)cur` when the television is cast into `String`
- It's also a pain to cast when you "**know**" a collection contains only a certain type
  - You still have to cast - to let the compiler know
- So collections of Object are not recommended

Lab 11.1: Using Collections

Other Collection Types

Overview

`List` and `ArrayList`

Autoboxing and Collections of Object

Other Collection Types

`Iterator`

More About Generics

The Collections Class

Set Interface

- **Set**: A collection containing **no duplicate elements**
  - Adds stipulation of no duplicate elements to `Collection` methods
  - Set interface does **not** add additional methods to Collection
- **SortedSet**: `Set` whose **elements are sorted**
  - Guarantees that iteration traverses the elements in sorted order
  - Adds **`first`** and **`last`** methods
- Concrete implementations of Set include:
  - `HashSet`, `LinkedHashSet`, `EnumSet`
  - `TreeSet` is a concrete implementation of `SortedSet`
- Sets are used similarly to lists
  - But duplicates won't be added in

Using Sets

- Below, we illustrate HashSet
  - Note how duplicates are not added
  - You can use a set to remove duplicates from another collection, as shown at the end of the example

```
// Create a HashSet
HashSet<String> set1 = new HashSet<String>();
String aString = "a";
String bString = "b";
set1.add(aString);
set1.add(bString); // set1.size()==2
set1.add(bString); // set1.size()==2 - duplicate not added

ArrayList<String> list1 = new ArrayList<String>();
list1.add(aString);
list1.add(bString);
list1.add(bString); // list1.size()==3 - duplicates added
                    // set2.size==2 - duplicates not added
HashSet<String> set2 = new HashSet<String>(list1);
```

Map Interface

- **Map**: Maps keys to values
  - Each key can map to at most one value
- Provides three views of the map
  - **key view** `Set` of keys (via `keySet()` method)
  - **entry view** `Collection` of values (via `values()` method)
  - **mapping view** `Set` of mappings (the collection itself)
- Common methods include:
  - `boolean` **`containsKey`**`(Object key)` : true if mapping for key exists
  - `boolean` **`containsValue`**`(Object val)`: true if a key maps to val
  - `V` **`get`**`(Object key)`: Returns value for the key
  - `Set<K>` **`keySet`**`()`: Returns set view of the keys
  - `V` **`put`**`(K key, V value)`: Associate key with value
  - `Collection<V>` **`values`**`()`: Returns collection view of values

HashMap

- `Map` implementation supporting efficient lookup of a value by a specified key
  - Key objects must have **`equals`** and **`hashCode`** methods

- `HashMaps` automatically ***rehash*** as needed
  - Creating a map with twice as many buckets
    - All elements are inserted into the new map and the old is discarded
  - Substantial processing overhead
    - A tradeoff between unused space versus processing time

HashMap<K,V>

- HashMap has two type parameters - `HashMap<K,V>`
  - `K` is the key type
  - `V` is the value type
  - Below we create a `HashMap` with a `String` key and `Television` value

```
HashMap<String, Television> floorSamples =
                    new HashMap<String, Television>();
```

  - `HashMap` has constructors to set initial capacity and load factors

- Generally use `put()` to add in values and associated key

Creating and Using HashMap

- Below, we creates a `HashMap` with an initial capacity
  - We add `Television` instances and `String` keys that represent the brand
  - Finally we retrieve one television based on the brand

```
// Create a HashMap of String key and Television value
HashMap<String,Television> floorSamples =
                    new HashMap<String,Television>(3);

// Add in Television samples keyed by brand
floorSamples.put("RCA", new Television("RCA",10));
floorSamples.put("Hitachi", new Television("Hitachi",10));
floorSamples.put("Sony", new Television("Sony",10));

// Get a television using its brand as a key
Television sonyTV = floorSamples.get("Sony");
```

Iterating Through a HashMap

- Iterating through a HashMap is a bit complicated
  - It has both keys and values
  - You can get the keys as a set, using `keySet()`
    - Brand strings in our example
  - You then iterate over the keys, getting the associated value
    - A television in our example

```
HashMap<String,Television> floorSamples =
                   new HashMap<String,Television>(3);
// Add in some elements (not shown)
Set<String> keys = floorSamples.keySet(); // Get the keys
// Iterate over the keys, and get each value
for (String curKey : keys) {
   Television curTV = floorSamples.get(curKey);
   System.out.println(curTV);
}
```

Lab 11.2: Using Sets

In this lab, you will create and populate both sets and lists

Iterator

Overview

`List` and `ArrayList`

Autoboxing and Collections of Object

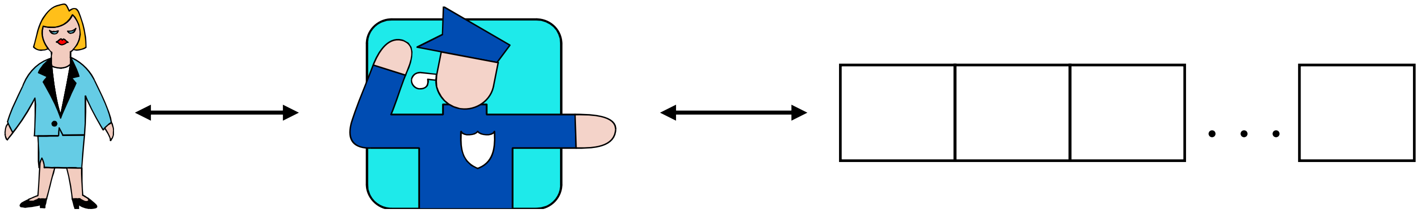
Other Collection Types

`Iterator`

More About Generics

The Collections Class

Processing Items with an Iterator



- Iterator: An object that steps through a collection's elements
  - The iterator deals with the collection details on your behalf
  - You interact with it and it interacts with the collection
  - Useful where the number of items **can't be determined**, or is **variable with respect to time**

- Using **for-each** is preferred when possible
  - But some libraries return iterators rather than collections
  - `Iterator` is also useful to know for backward compatibility

Iterator Interface

- `java.util.`**`Iterator`** defines the methods for iterating over (traversing) a collection's values
  - `Collection.iterator()` provides access to an iterator
  - `iterator()` returns an iterator over the elements in its collection

```
Iterator<E> iterator()
```

- `Iterator` methods:
  - `boolean` **`hasNext`**`()`: Move forward in the collection
    - Returns `true` if another element exists, else `false`
  - `E` **`next`**`()`: Return the next element in collection.
  - `void` **`remove`**`()`: Remove the last element returned

Using Iterator - Example

- Below, we get an iterator from a collection of `Television`
  - We use `hasNext()` and `next()` to iterate through the collection and retrieve the elements
  - It works fine, but `for-each` is shorter and cleaner

```
// Declare collection of Television
Collection<Television> tvs = new ArrayList<Television>();

// Add in instances somehow …

Iterator<Television> i = tvs.iterator(); // Get the iterator
while (i.hasNext()) {             // While there are elements left
  Television tv = i.next();       // Get the element
  tv.mute();
}
```

[Optional] More About Generics

Overview
`List` and `ArrayList`
Autoboxing and Collections of Object
Other Collection Types
`Iterator`
More About Generics
The Collections Class

What Are Generics

- ***Generics***, support types and methods that work with any type
  - A **type parameter** specifies the type

- We saw this with collections - `ArrayList`**`<Television>`**
  - Previously, collections held Objects, with methods like **`add(Object o)`**,
  - Not type safe and required casting
  - e.g. Adding an `Integer` to a collection of `String`, would compile, but eventually fail at runtime

- Generic collections add type safety, with no need for casting
  - The add method is defined as **`add(E o)`**
  - E is a type parameter – can only add elements of type E

Declaring a Generic Class

```
// Example of Collection interface definition - most detail omitted
public interface Collection<E> extends Iterable<E> {
  int size();                    // Returns the size of the collection
  boolean contains(Object o); // Does the collection contain Object o
  boolean add(E o);              // Add o to the collection
}
```

```
// Example of ArrayList definition - most detail omitted
public class ArrayList<E> extends AbstractList<E> implements List<E> {
  private transient E[] elementData; // Buffer holding data

  public boolean add(E o) {
    ensureCapacity(size + 1);
    elementData[size++] = o;
    return true;
  }
}
```

- Above, we illustrate generic type definitions
  - For the most part, they look familiar
  - Except for the placeholder (in this case **E**) representing a type

Summary - Basic Generics Usage

- Generic types have **type parameters**
  - Type parameters are declared in their definition - e.g.
    `public interface Collection <E>`
  - Type parameters are useable within the entire class - similarly to a normal type - e.g.
    `public boolean add (E anElement)`
  - When using a generic type, you provide the type arguments
    `Collection<Television> = new Collection<Television>();`
  - Arguments to methods may be of a parameterized type, e.g.:
    `public void muteAll(Collection<Television> tvs)`

Using Generics - Example

```
// Declare type-safe collection Television (declared elsewhere)
Collection<Television> tvs = new ArrayList<Television>();
// Add in Television instances somehow …
for (Television tv : tvs) {  // Iterate
  tv.mute();          // Simple iteration, no casting !
}
```

```
// Generics: Compiler only allows Strings to be added to brands
ArrayList<String> brands = new ArrayList<String>();
brands.add("RCA");              // OK
brands.add(new Television());   // ERROR – will not compile
// ...
// we now have type-safe iteration, too
for (String brand : brands ) {
  String curBrand = brand;      // don't have to cast
}
```

- We show an example of generics above
  - The top example creates and uses a collection of `Television`
  - In the above example, we use a collection of `String` incorrectly
    - This would not compile

Inheritance with Generic Types

- You can inherit from or implement a generic type
  – Using either a specific parameterization, or carrying over the parameterization
- Below, we define a collection type that is also parameterized
  – It adds the method `showAll()` that uses the type parameter
  – At bottom, we show how it's parameterized when we use it

```
public class MyCollection<E> extends ArrayList<E> {
  public void showAll() {
    for (E cur : this) {
      System.out.println(cur);
    }
  }
}
```

```
MyCollection<Television> tvs = new MyCollection<Television>();
tvs.add(new Television("Sony", 10));
tvs.showAll();
```

Inheritance with Generic Types

- Below, we define a collection type that is not parameterized
  - It specifies the type parameter **in the extends**
  - At bottom, we show that you don't parameterize it when using it

```
public class TVCollection extends ArrayList<Television> {
  public void muteAll() {
    for (Television cur : this) {
      cur.mute();
    }
  }
}
```

```
TVCollection tvs = new TVCollection();
tvs.add(new Television("Sony", 10));
tvs.add(new Television("RCA", 5));
tvs.muteAll();
```

Assignment with Generic Types

- Assignment of generic types with the **same** type argument works fairly normally
  - Below, we assign an `ArrayList` to a `Collection` ref –
  - OK, since they're both parameterized by `String`

```
Collection<String> c = new ArrayList<String>();
```

- Assignment of generic types with **different** type arguments is **not** allowed - the below is an error

```
ArrayList<Object> c = new ArrayList<String>();
```

- You can assign to raw types
  - You'll get a compiler warning since it can lead to run-time errors

```
Collection c = new ArrayList<String>();
```

Wildcard Parameter Types

- Generic types can have wildcard parameters – denoted by **?**
  - This is only legal for references
    - It is not something that could appear in a new expression
- Below, `Collection<?>` stands for a family of types
  - `c` can refer to any parameterization of `Collection`

```
Collection<?> c = new ArrayList<String>(); // OK
c = new ArrayList<Television>; // Also OK
```

- It can get complex, e.g. the following
  - `Collection<? extends Television>`
  - `Collection<? super Television>`
  - Details are beyond the scope of the course

Generic Methods

- Methods may be parameterized individually
  - Without parameterizing the enclosing class
  - Declare a type parameter at the beginning of the method
- `nCopies()` below returns a list containing n copies of the passed in object
  - <T> is a type parameter
  - Note how T is used in the return and argument types

```
public static <T> List<T> nCopies(int n, T o)
```

- Method parameters can also be wildcarded, as shown below

```
public static <T> void copy(List<? super T> dest,
  List<? extends T> src)
```

[Optional] The Collections Class

Overview

`List` and `ArrayList`

Autoboxing and Collections of Object

Other Collection Types

`Iterator`

More About Generics

The Collections Class

Collections Class

- Contains static methods operating on collections
  - Functionality is useful for all collections
  - Two main areas of functionality
- **Wrapper methods**: Return a new collection backed by an existing collection - changes behavior in three areas:
  - **Synchronization wrappers**: Add automatic thread-safety
  - **Unmodifiable wrappers**: Remove ability to add/remove element
  - **Checked interface wrappers**: Provide dynamic (run-time) type checking of elements added to a collection
- **Algorithms**: Methods operating on collections
  - Sorting, shuffling, routine data manipulation, searching, composition, and finding extreme values

Unmodifiable Wrappers

- Disallow modification of the collection
  - Intercepts all operations that modify the collection
  - e.g. `add()`
  - The "modifying" methods will throw an exception
- Different methods are provided for different capabilities
  - And different versions for many kinds of collections (e.g. sets)
  - We show some below - see the `Collections` JavaDoc !
  - Note the use of generics (you'll see this a lot)

```
public static Collection<T>
       unmodifiableCollection(Collection<? extends T> c)

public static List<T>
       unmodifiableList(List<? extends T> list)
```

Unmodifiable Example

- Below, calling `add()` throws an exception
  - **`UnsupportedOperationException`**
  - Calling any "modifying" method will throw an exception
  - Lets you be **absolutely sure** a collection remains unmodified
  - Just pass a reference to the wrapper

```
ArrayList<String> normal = new ArrayList<String> ();
normal.add("A string");
normal.add("Another string");

List<String> unmodifiable =
    Collections.unmodifiableList(normal);
unmodifiable.add("Does this work");
```

Checked Interface Example

- Below, the second `rawCollection.add()` throws a run-time exception
  - The first call compiles, but gives problems later
  - See notes if you want more detail

```
ArrayList<Date> normal = new ArrayList<Date> ();
normal.add(new Date());
normal.add(new Date());

List<Date> checked =
      Collections.checkedList(normal, Date.class);
Collection rawCollection = normal;
rawCollection.add("Fine now, but give problems later");
rawCollection = checked;
rawCollection.add("This will throw an immediate exception");
```

Algorithms

- Provide reusable pieces of collection functionality
  - Static methods in `Collections`
  - Most defined on `List`, some on `Collection`
  - We show several below, and an example on the next slide

```
// Search for the specified object using binary search
public static <T> int binarySearch(
                 List<? extends Comparable<? super T>> list, T key)

// Number of elements in collection equal to the specified object
public static int frequency(Collection<?> c, Object o)

// Return true if collections have no elements in common
public static boolean disjoint(Collection<?> c1, Collection<?> c2)

// Sort specified collection in ascending order
public static <T extends Comparable<? super T>> void sort(
                                                    List<T> list)
```

Sort Example

- Below, we show a `sort()` example - note its simplicity
  - Note: Algorithms **may change the source collection**

```
ArrayList<String> fruit = new ArrayList<String> ();
fruit.add("Orange");
fruit.add("Apple");
fruit.add("Lemon");

System.out.println(fruit); // Outputs [Orange, Apple, Lemon]

Collections.sort(fruit);
System.out.println(fruit); // Outputs [Apple, Lemon, Orange]
```

Review Questions

1. List some differences between arrays and collections.
2. What is the root interface of the collections hierarchy?
3. What are two ways to traverse a collection - which is better?
4. What is the fundamental difference between a list and a map?
5. What are generics, and why are they useful?
6. How do the collections use generic types?

Lesson Summary

- Collections represent **a group of objects** that supplement arrays
  - They can grow at runtime
  - They have more capabilities than arrays
    - e.g. sets with no duplicates and maps of keys to values
  - They can't store primitives,
    - But can store "wrapped" values (e.g. `Integer`)

- All collection types derive from the **`Collection`** interface
  - Defines basic operations
    - Adding/removing elements
    - Getting the size
    - Converting to an array
  - Many collection types that support different needs

Lesson Summary

- Traverse collections using **`for-each`**
  - Or get an iterator - more verbose, but some more capability and control

- Several collection types, including:
  - **`List`**: Ordered list of objects accessible by index
  - **`Set`**: Collection containing no duplicates
  - **`Map`**: Collection mapping keys to values accessible by key

- Generics provide **compile-time type safety**
  - They support typed collections, e.g. **`ArrayList<String>`**
  - Use generic collections wherever possible
  - They're safer and easier

Session 12: Database Access with JDBC and JPA

JDBC Overview

JPA Overview

Working with JPA

Session Objectives

- Explain JDBC's role in database access
- Describe the issues of object-relational mapping
- Describe the overall goals of the Java Persistence API (JPA)
- Describe the JPA architecture
- Create a simple JPA application
- Map a simple class to a DB using JPA
- Be introduced to updating and inserting persistent objects
- Be introduced to the Java Persistence Query Language (JPQL)

JDBC Overview

JDBC Overview

JPA Overview

Working with JPA

What is JDBC?

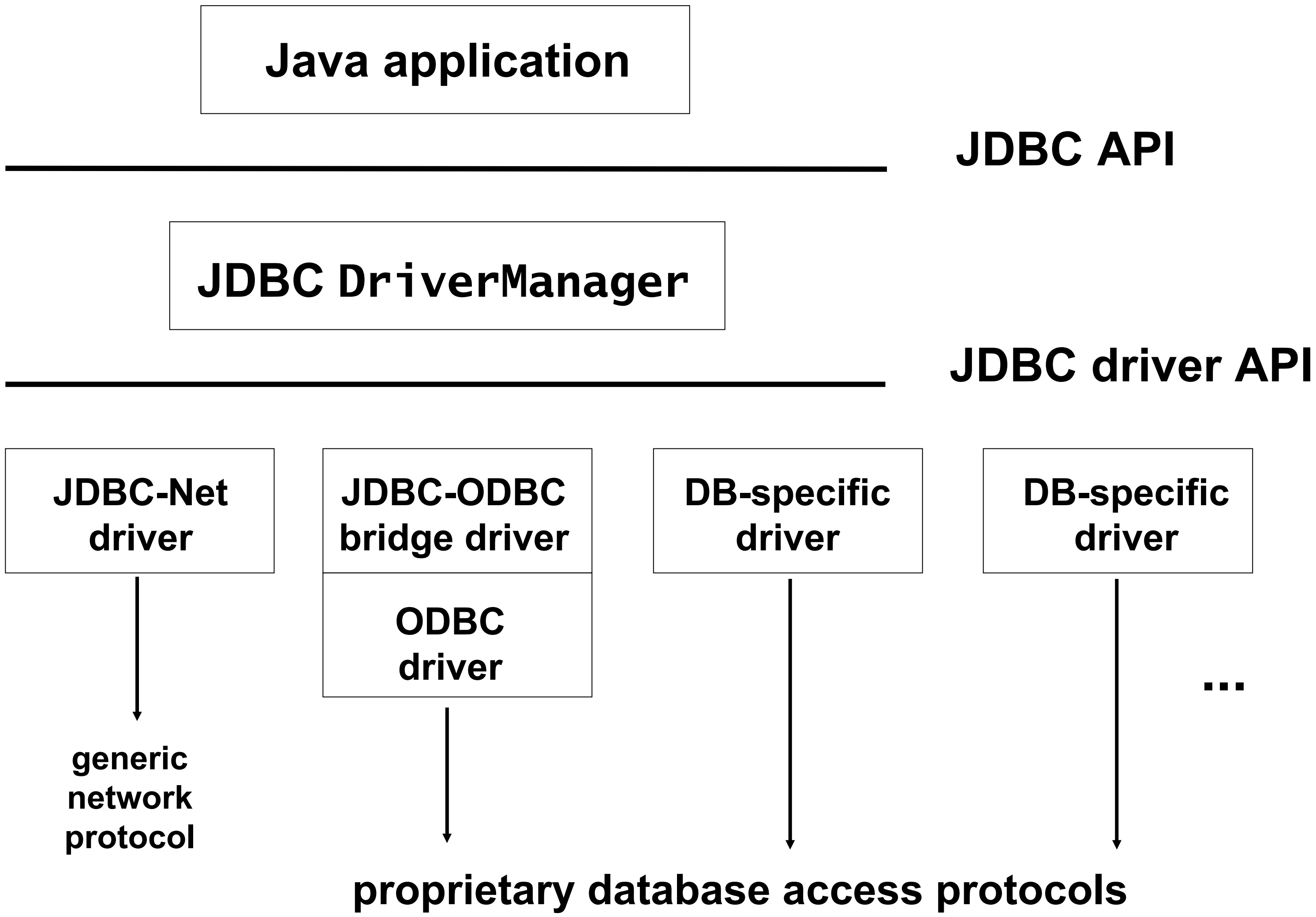
- **JDBC** provides connectivity to a relational DB
  - Standard API included with the JDK

- Generic SQL database access framework for Java
  - Call-level SQL interface for Java
  - Executes raw SQL statements and retrieves the results

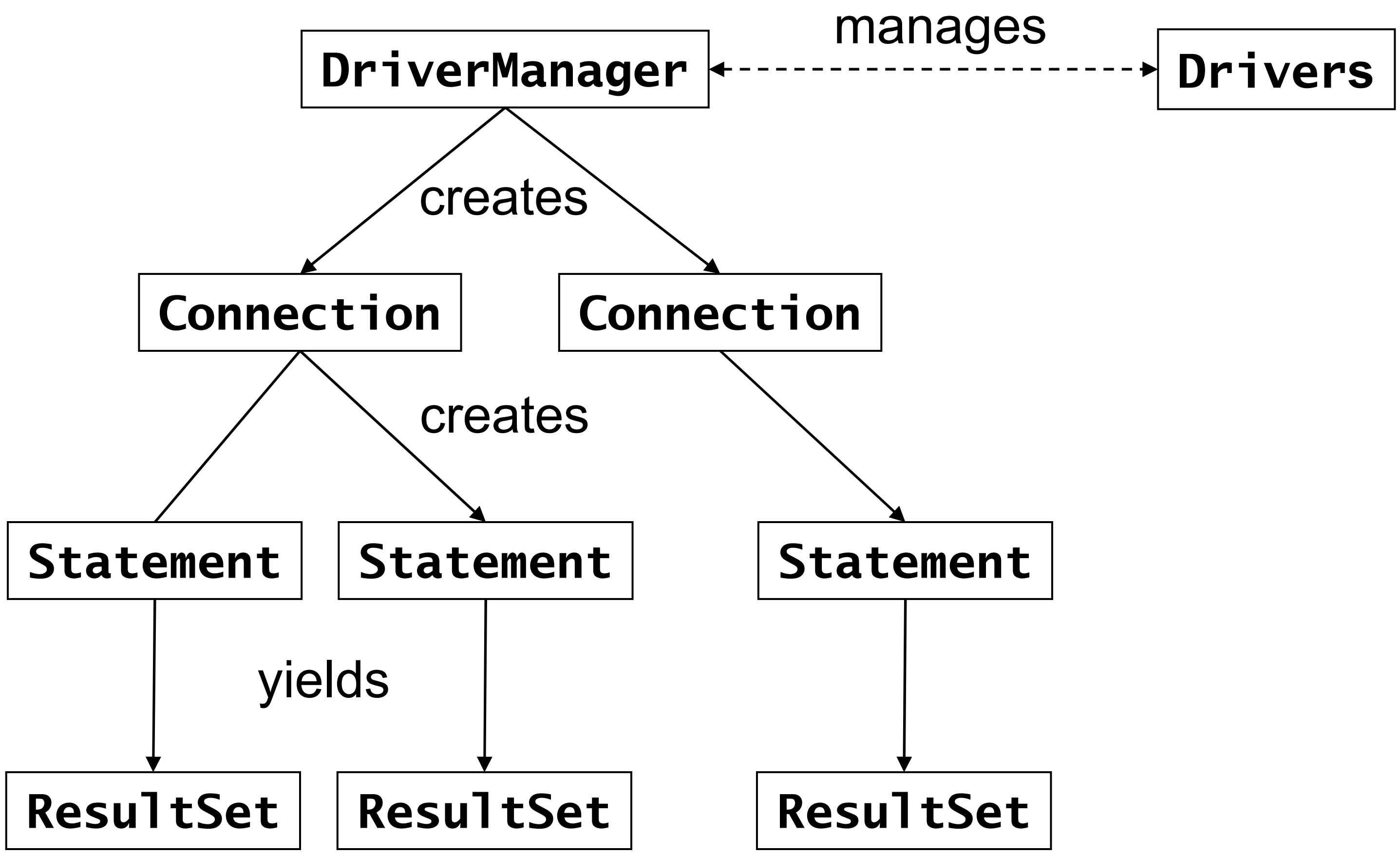
- **JDBC 4.2 is part of Java 8**

- Consists of two packages:
  - `java.sql` the core JDBC API
  - `javax.sql` Enterprise capabilities (e.g. DataSource)
  - Both packages are now part of the Java JDK

JDBC Architecture



The Fundamental JDBC API



JDBC instantiation diagram (Factory Pattern)

Common JDBC Types

- `DriverManager`: Factory for database connections
  – Also manages database drivers
- `Connection`: Represents a session with a specific database
  – Factory for statements
- `Statement`: Represents dynamic queries
- `PreparedStatement`: Represents precompiled queries
- `ResultSet`: Represents a tabular set of data generated by a `Statement`
- We'll show a simple JDBC (but not in depth)
  – JPA is a better solution for DB access
  – If you need JDBC, see the JDBC appendix

Naming Databases with URLs

- A JDBC URL locates a DB for JDBC
  - Syntax: **`jdbc:<subprotocol>:<subname>`**

- **`<subprotocol>`**: Specifies how you connect to the database
  - For example, oracle, db2, sybase, derby, odbc
  - We'll illustrate `derby`, the DB used in the labs

- **`<subname>`**: Identifies a particular database
  - In our labs, it will be `//localhost:1527/JavaTunesDB`
  - The format of this name is determined by the database driver

- For our labs, the complete JDBC URL will be:
  - **`jdbc:derby://localhost:1527/JavaTunesDB`**

The Item Database Table

- The labs are based on the online JavaTunes music store
- A table to store music items is shown below (1)
  - We'll use this in the labs
  - Note that the **table name is `Item`**
  - Note the primary key (in `ITEM_ID`) **is an Identity column**
  - Note the **names and types of the other properties**

```
CREATE TABLE Item
(
   ITEM_ID     BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY (START
                                             WITH 1, INCREMENT BY 1),
   Title       VARCHAR(40),
   Artist      VARCHAR(40),
   Price       DECIMAL(5,2),
   CONSTRAINT  PK_Item  PRIMARY KEY(ITEM_ID)
);
```

Database Connection - Example

- Below, we get a DB connection with DriverManager
  - The DB "name" is **jdbc:derby://localhost:1527/JavaTunesDB**
  - We use `try-with-resources` to close the connection
  - The next slides show examples of querying with JDBC directly

```
// Code fragment

// get a connection to the database using try-with-resource
try (Connection conn =   DriverManager.getConnection(
  "jdbc:derby://localhost:1527/JavaTunesDB",
  "guest", "password"); ) {

  // rest of code
}
catch (SQLException sqle) {
  System.out.println(sqle);
}
// No finally or close of connection needed due to
// use of try-with-resources
```

Using Statement - Example

```
// Define the SQL
String sql = "SELECT ITEM_ID, Title, Price FROM ITEM";
try {    // conn is connection as created earlier

  // create a Statement object
  Statement stmt = conn.createStatement();

  // execute the SQL with it
  ResultSet rs = stmt.executeQuery(sql);
  while (rs.next())
  {
    long       id    = rs.getLong("ITEM_ID");
    String     title = rs.getString("Title");
    BigDecimal price = rs.getBigDecimal("Price");
  }  // Do something with data (not shown)
}
catch (SQLException sqle) {
  System.out.println(sqle);
}
```

Using PreparedStatement - Example

```
// define the ?-SQL (for PreparedStatement)
String sql = "SELECT Title, Price FROM Item " +
                "WHERE ITEM_ID = ?";

try {  // conn is connection as created earlier

  // prepare the ?-SQL with the DBMS PreparedStatement
  PreparedStatement pstmt = conn.prepareStatement(sql);

  // set the ? - they are numbered starting at 1
  pstmt.setInt(1, 1);

  // execute the PreparedStatement and get a ResultSet
  ResultSet rs = pstmt.executeQuery();
  // Process as before
}
catch (SQLException sqle) {
  System.out.println(sqle);
}
```

Summary

- JDBC is Java's foundation for database access
  - Handling database connections, queries, processing results
- JDBC is low level and cumbersome to use directly
  - A single query requires many steps
  - Processing the results requires even more work
    - e.g. packaging the data up so it's easy to use
- We'll look at **JPA** (Java Persistence API) next
  - It handles a lot of the low-level code for DB access
  - Making it much easier, requiring much less coding
- There is an appendix with more JDBC if its needed

JPA Overview

JDBC Overview

JPA Overview

Working with JPA

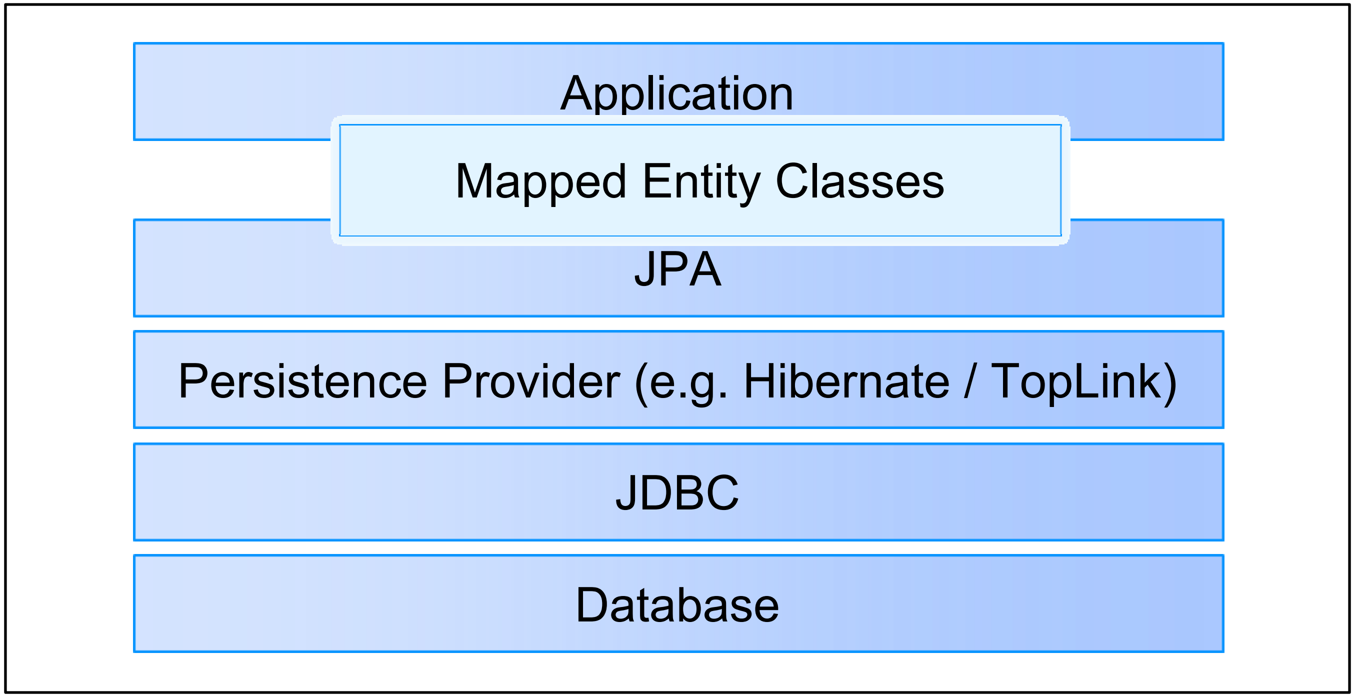
Java Persistence API (JPA) Overview

- Addresses object-relational mapping (ORM) issues
  - **OO Model**: Interacting objects traversed via relationships
  - **Relational Model**: Tables joined via foreign key relationships
  - Going between these models requires effort

- **JPA**: Standard persistence framework for Java, goals include:
  - Provide **ORM capability** to use a **Java OO domain model** to work with relational data
  - Provide a **light-weight POJO based persistence framework**
  - Provide a **query language** to **remove or encapsulate vendor specific SQL** code
  - **Relieve the developer from 95%** of persistence programming

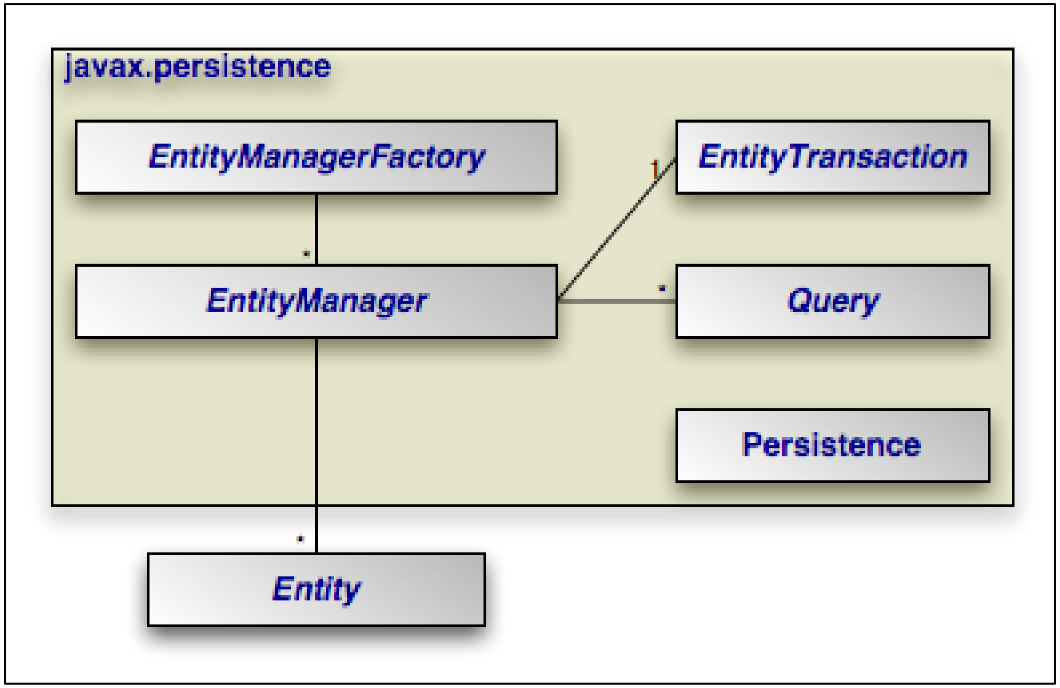
JPA Architecture – High Level View

- JPA uses entity **metadata** to provide persistence services
  - Generally, implementations are built over an existing persistence provider, e.g. Hibernate
  - They also generally use JDBC



JPA Architecture – Programming View

- Key JPA types include:
  - **Persistence**: For configuration of the system
  - **EntityManager**: Manages a set of persistent entities
  - **EntityManagerFactory**: Factory for EntityManagers
  - **Query**: For finding entities
  - **Entity**: POJO class mapped using JPA



Working with JPA

JDBC Overview

JPA Overview

Working with JPA

Entity Classes

- Entity: A **lightweight persistent domain object**
  - Represents data stored in a DB
- Entity **metadata** describes the mapping to the data store
  - With annotations (preferred) or XML
  - Core annotations are in **`javax.persistence`**
- Entities must:
  - Have a no arg constructor (so JPA can instantiate it)
  - Contain fields to hold persistent state
    - Must **NOT** be declared public
    - Clients use get/set methods to access data
  - Provide an identifier (usually called id - maps to DB primary key)

MusicItem Entity Class

- `com.javatunes.persist.MusicItem` is used in JavaTunes
  – Represents an item of music, and is defined below (1)
- `@Entity` declares the class to be a persistent entity
- `@Id` denotes that this field holds the primary key
- Fields without annotations are mapped by default
  – To a column with the name of the property, e.g. `title`

```
// Much detail omitted - package, constructors, …
import javax.persistence.*;

@Entity
public class MusicItem {
   @Id
   private Long       id;
   private String     title;
   private String     artist;
   private BigDecimal price;

   public MusicItem() { /* … Detail omitted … */ }
}
```

Annotations in Brief

- Adds metadata to your source code
  - A **rubber stamp** usable on fields, methods, types, etc
  - Read by tools (e.g. JPA runtime)
  - Syntax is a bit odd - **`@NameOfAnnotationType`**
  - More on them later

- Think of an envelope stamped **Fragile** and **Next Day**
  - This doesn't change the envelope or its contents
  - May change how the envelope is processed

- Similarly, JPA annotations affect how JPA processes entities
  - JDBC/SQL is generated based on the annotations
  - In the end, saving you a lot of work !!

Additional MusicItem Annotations

- We need to specify the table name (since it's not MusicItem) (1)
  – **@Table** specifies the DB table for an entity class
- The id field is stored in the DB ITEM_ID column
  – **@Column** can specify a DB column name
- Our DB primary key is an identity column (2)
  – **@GeneratedValue(strategy=GenerationType.IDENTITY)**
  – Specifies we're using an identity column in the DB

```
// Much detail omitted - imports, package, constructors
@Entity
@Table(name="ITEM")
public class MusicItem {
   @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="EVENT_ID")
   private Long      id;
} // Other properties omitted
```

Lab 12.1: Mapping an Entity Class

In this lab, we will map a class to a database using JPA annotations - we won't access the DB yet

The Persistence Unit

- **Persistence unit**: Defines the set of entities managed by an entity manager (covered next)
  – Plus DB connection info and other info

- The classes in a persistence unit include:
  – Annotated entities (may be in jar files)
    - Either auto-detected (scanned) or listed in *persistence.xml* (1)
  – Can also be mapped in XML *orm.xml* (2)

- A persistence unit is configured in an XML config file
  – Generally called ***persistence.xml***
    - Should be under the META-INF folder
  – Supports DB connection info, and other global configuration
  – It's the central configuration file for JPA

persistence.xml Structure

- **<persistence-unit>** defines the persistence unit
  - **name** attribute: Required, specifies the persistent unit name
  - **transaction-type** attribute: Optional, specifies transaction type
    - Resource local (i.e. JDBC transactions) used here – suitable for Java SE *
  - **<properties>** holds config properties for the provider
    - e.g. the **hibernate.dialect** property below
    - Tells Hibernate what database is being used (1)
    - We'll look at more properties in the lab for our environment

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="…" version"2.0"> <!-- namespaces not shown -->
   <persistence-unit name="javatunes"
                     transaction-type="RESOURCE_LOCAL">
      <properties>
        <property name="hibernate.dialect"
             value="org.hibernate.dialect.DerbyTenSevenDialect"/>
      </properties>
   </persistence-unit>
</persistence>
```

The EntityManager

- **`EntityManager`** (or **EM**): Used to interact with a database, to create, read, or write an entity
  - It also **manages** entity instances

- Important `EntityManger` methods include:
  - **`<T> T find(Class<T> entityClass, Object primaryKey)`**: Find an entity by its primary key (obscure syntax - don't worry)
  - **`void persist(Object entity)`**: Make a new entity instance managed and persistent
  - **`void refresh(Object entity)`**: Refresh instance state from DB instance from the DB, overwriting changes made to the entity
  - **`void remove(Object entity)`**: Remove the entity instance

EntityManager Factory and EM Creation

- Create an EM with an **EntityManagerFactory** (**EMF**)
  - Obtain an EMF via the `Persistence` class and the name of the persistence unit in *persistence.xml*
- Below we get an EMF using the `javatunes` persistence unit
  - We get an EM from it
  - We close the EM and EMF when done to release their resources

```
// Get an entity manager factory
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("javatunes");
// Get an entity manager
EntityManager em = emf.createEntityManager();
try {
   // Do persistence related work here
 }
finally {
  em.close();  // Close the entity manager
  emf.close(); // Close the entity manager factory
}
```

Working with Transactions

- Java SE apps generally use a **resource-local** entity manager
  - Directly controlling the TX boundaries with the JPA API
  - Configured in *persistence.xml*:

```
<persistence-unit name="javatunes"
                  transaction-type="RESOURCE_LOCAL">
```

- Manage the TX via a **`javax.persistence.EntityTransaction`** object obtained from the EM
  - Below, we also use `EntityManager.find()` to retrieve an item by id
  - Note that we pass `MusicItem.class` as an argument to find
  - Indicates that the generic `find` method returns an instance of `MusicItem`

```
EntityManager em = // Initialization as shown previously
em.getTransaction().begin();  // Begin a transaction
MusicItem m = em.find(MusicItem.class, new Long(1));
em.getTransaction().commit();  // Commit a transaction
```

Complete JPA Example

- Below we show all the pieces together

```
// Get an entity manager factory
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("javatunes");
// Get an entity manager
EntityManager em = emf.createEntityManager();
try {
  em.getTransaction().begin();  // Begin a transaction
  MusicItem m = em.find(MusicItem.class, new Long(1));
  em.getTransaction().commit();  // Commit a transaction
}
finally {
  em.close();  // Close the entity manager
  emf.close(); // Close the entity manager factory
}
```

Summary

- Standard steps for using JPA
  - Get an `EntityManagerFactory`
    - Using the `Persistence` class
  - Get an `EntityManager` from the `EntityManagerFactory`
  - Start a TX using the `EntityManager`
  - Do your work, e.g. do a `find()`, and any other work
  - Finalize the TX, and close all resources
  - In a Java EE environment, much of the boilerplate code is different (1)

- It all this worth it?
  - It looks complicated, and may be obscure to you right now
    - Interfaces, exceptions, delegation, etc.
  - It saves a LOT of work over plain JDBC/SQL
    - Once it's set up, and you're comfortable with it

Lab 12.2: Using JPA

In this lab, we will use the JPA EntityManager

Persisting a New Entity

- It's easy to insert new instances (rows) into the DB
  - Create a new (transient) instance using *new*
  - Set values for the properties - except, generally, the ID [(1)]
  - Save the instance to the DB via `EntityManager.persist()`

  `void persist(Object entity)`

- The instance is inserted into the DB
  - Once the TX commits, it is a full-fledged (managed) entity and appears in the database

```
try { // Entity manager (em) initialization not shown
  em.getTransaction().begin();  // Begin a transaction
  MusicItem newItem = new MusicItem();
  newItem.setArtist("Madonna");
  newItem.setTitle("Material Girl");
  em.persist(newItem);
  em.getTransaction().commit();  // Commit a transaction
}
```

Updating a Persistent Instance

- You can easily update a persistent (or managed) instance
  - This is one currently associated with a persistence context
  - Update the instance, and the changes are persisted
  - JPA detects any changes and **synchronizes the state with the database** when the TX completes

```
// Assume the code fragment occurs in a transaction context and
// you have an initialized EntityManager reference (em)

Long itemId = new Long (5); // Assume this is the id we want
MusicItem m = em.find(MusicItem.class,itemId);

// Change will be automatically persisted
m.setTitle("Amazing Song Title");

// When Tx commits, the changes are persisted to database
```

Removing an Instance

- It's easy to delete a managed instance from the DB
  - Just call **remove()** on the instance
  - When the context synchronizes with the DB, the row is deleted
  - Note that rows are often **never deleted** in production systems
    - More commonly, old data is kept for historical queries

```
// Assume a transaction, and EntityManager reference, as before

MusicItem m = em.find(MusicItem.class, new Long(5));

// Remove the item
em.remove(m);

// When Tx commits, the deletion is persisted to database
```

Executing a Query

- Execute queries via the **Query** interface
  - Obtain a `Query` instance from the EM
- Write queries in **Java Persistence Query Language** (**JPQL**)
  - An OO query language that is part of JPA
- Below, we retrieve all items where the artist is Madonna
  - Note how simple this is
  - More details are beyond the scope of this brief introduction

```
// Assume that em is an initialized EnityManager reference
TypedQuery<MusicItem> q = em.createQuery(
    "SELECT mi FROM MusicItem mi WHERE mi.artist = 'Madonna'",
    MusicItem.class);
List<MusicItem> resultList = q. getResultList();
for (MusicItem cur : resultList) {
  System.out.println(cur);
}
```

Lab 12.3: Insert/Query Demo

In this lab, we will demonstrate some additional features of JPA

Review Questions

- What is JDBC?
- What does the Java Persistence API (JPA) do, and why do we need it?
- What is an entity?
- How do you map an entity to the database?
- What is the id property of an entity?
- What does the entity manager do?
- How are queries defined with JPA?

Lesson Review

- **JDBC** is a low-level SQL-based Java API for accessing relational databases
  - Provides access to the database, statements, querying, etc. .
- **JPA** (Java Persistence API) is a standard Java object-relational mapping (ORM) framework
  - It maps Java objects to a relational DB
  - Built on top of JDBC, it handles the details of generating JDBC code, SQL, and converting to/from Java objects and DB tables
- JPA is POJO-based
  - Map persistent entities via annotations
    - Or use common defaults
  - Every entity has an id which uniquely identifies it in the DB

Lesson Review

- An **`EntityManager`** stores / retrieves entities from the DB
  - An instance interacts with a specific database
  - It is configured in a persistence unit (*peristence.xml*) for that DB
  - It provides methods to store and query entities

- Queries are defined in **JPQL**
  - Java Persistence Query Language
  - Queries are executed via the **`Query`** type
  - Results are retrieved using the **`ResultList`** type

Session 13: Additional Language Features

Assertions

Annotations

Lambda Expressions

Other Java Features

Session Objectives

- Understand what assertions do and know when and how to use them
- Understand the concepts of Java Annotations
- Be familiar with lambda expressions
- Learn about additional features in Java

Assertions

Assertions

Annotations

Lambda Expressions

Other Java Features

Assertions Defined

- Statements letting you check assumptions in code
  - Contain a `boolean` expression you believe will be `true`
  - If it's **not** `true`, an **`AssertionError`** is thrown
  - This is considered a failure (bug) in the code
    - You generally don't catch and handle it
    - You need to track it down and fix your code
    - Different from an application exception like setting the volume of a `Television` to a negative number
- A built-in and better alternative to manual checking/debugging with `if` statements
  - Assertion syntax is concise and it's more flexible
  - Assertions are easily and flexibly enabled/disabled at runtime

Assertion Uses

- Common uses include checking:
  - **Preconditions** – must be satisfied at the start of a method
  - **Postconditions** – must be satisfied at the end of a method
  - **Invariants**
    - **Internal/State** – check that data values and structures are correct
    - Replace comments like "we know volume >=0" with assertions
    - **Flow of control** – check that control does not flow unexpectedly
    - Replace "Should never get here" comments with assertions

Assertion Non-Uses

- **Checking parameters in public methods**
  - Parameter checking is part of a method's contract
    - It should always perform important parameter checks
  - If assertions are disabled, the checks are gone
  - Assertion failures only throw `AssertionError`
  - They can't throw something more appropriate like `IllegalArgumentException`
- **Performing any required work for the application**
  - If assertions are disabled, the required work is not performed
- **If the boolean-expression causes side-effects**

Assertion Syntax

- **`assert`** statements have two forms:

  `assert` *boolean-expression* **`;`**

  `assert` *boolean-expression* **`:`** *any-expression* **`;`**

- If *boolean-expression* is `false`, an **`AssertionError`** is thrown
  - In the first form, the `AssertionError` has no detail message
  - Use the second form to provide a detail message
    - The runtime system passes the value of *any-expression* to the `AssertionError` constructor
    - Generally not an end user message - should help the developer to diagnose/fix the error causing the assertion failure

Using Assertions to Check State - Example

- Below, is an example based on television muting
  - We know the volume should be zero after muting
  - This is a good use of assertions
  - We include a detail message in this case

```
public void mute() {
  if (this.isMuted())        // currently muted, volume = 0
  {
    assert this.getVolume() == 0 :
      "Television is muted but volume=" + this.getVolume();
  }
  else
  {
    // ...
  }
}
```

Using Assertions to Check Flow of Control

- A `switch` statement with no default case is often making an assumption – that one of the cases will always be executed

```
switch (direction) {   // char direction is 'N','S','E','W'
  case 'N':
    ...
    break;
  case 'S':
    ...
    break;
  case 'E':
    ...
    break;
  case 'W':
    ...
    break;
  default:
    assert false : direction;
    // OR throw new AssertionError(direction); (see notes)
}
```

Enabling/Disabling Assertions at Runtime

- By default, assertions are **disabled** at runtime
  - Any performance penalty is removed
  - The assertions are equivalent to empty statements
  - But they're still present in the *.class* files

- To enable them, use **`java -enableassertions`** **or** **`–ea`**
- To disable them, use **`java -disableassertions`** **or** **`-da`**
  - The flags can take an additional argument
    - **no argument** enabled for all classes
    - **`package-name...`** enabled for classes in named package and its subpackages (the `...` is part of the syntax)
    - **`...`** enabled for classes in unnamed package in current directory
    - **`class-name`** enabled for that class

Enabling/Disabling Assertions - Examples

- The enable/disable flags are processed left-to-right and are cumulative

```
enable assertions globally
java -ea TelevisionTest

enable assertions only for package com.entertainment
java -ea:com.entertainment... TelevisionTest

enable globally, but disable for package com.entertainment
java -ea -da:com.entertainment... TelevisionTest

enable for package com.entertainment, except for Radio
java -ea:com.entertainment... -da:com.entertainment.Radio
  TelevisionTest
```

What They Look Like at Runtime

- Using our previous television example
  - We enable assertions by running it as:

```
java -ea TelevisionTest
```

  - We show you the output at bottom where the assertion failed

```
public void mute() {
  if (this.isMuted())        // currently muted, volume = 0
  {
    assert this.getVolume() == 0 :
      "Television is muted but volume=" + this.getVolume();
  }
  else
  ...
}
```

```
Television: brand=RCA, volume=10
Television: brand=RCA, volume=<muted>
Exception in thread "main" java.lang.AssertionError: Television is muted: true, volume=7
        at Television.mute(Television.java:61)
```

Annotations

Assertions

Annotations

Lambda Expressions

Other Java Features

The Issue

- Many technologies rely on "side files" that need to be consistent with the "base file" (which is the Java source code)

base file	side file(s)
JavaBean class	BeanInfo class
JEE class	deployment descriptor
Web Service class	WSDL file

- Modifying the Java source often requires keeping the side file(s) consistent with the modification
  - Many side files are hand-edited XML documents
  - This is error-prone

Annotations - The Solution

- Mechanism for adding metadata to your source code
  - For use by tools, which read the source metadata
  - They can then add functionality or generate the side file - nice!
  - Generally use **`apt`** – Java's **annotation processing tool**
- You can both **declare** and **use** annotation types
  - Using **@** to denote something is annotation related
- JavaDoc uses similar syntax and architecture
  - Javadoc comments (`/** ... */`) are read by the **`javadoc`** tool
  - It generates HTML-based API documentation
  - Javadoc comments use **`@something`** syntax also
    - Used to add information to the generated documentation

Using Annotations Example

```
public class Television implements Volume {
   @Override
   public String toString() { /*…*/ }
}
```

```
@Stateless
public class CatalogBean implements Catalog {
   public Collection findByKeyword(String keyword) { /*…*/ }
}
```

- `@Override` indicates that a method overrides a supertype method
  - It's a compilation error if you are not overriding [1]
- `@Stateless` illustrates an annotated EJB 3 declaration
  - apt generates the entries for the EJB deployment descriptors
- It's extensible - You can write your own annotations and even your own annotation processor

Lambda Expressions (Java 8)

Assertions

Annotations

Lambda Expressions

Other Java Features

Motivation: Common Actions are Verbose

- Previously, Java was **verbose** in many situations
  - e.g. requiring a class definition for one method
- For example, in the Swing GUI code below (1)
  - A button's action is supplied by defining an `ActionListener` implementation
  - Result: We **defined a new class** just for one method
  - You can make this less verbose, but it's still cumbersome (2)

```
public class MyListener implements ActionListener {
  public void actionPerformed(ActionEvent ae) {
    System.out.println("Button Pressed");
  }
}
```

```
// Code fragment using MyListener
JButton submitButton = new JButton("Press Me");
sumbitButton.addActionListener(new MyListener());
```

Solution: Lambda Expressions

- Lambda expressions **encapsulate a single unit of behavior**
  - e.g. - the single method in `ActionListener`
  - You can pass that behavior e.g. to our button
  - We'll provide a brief overview here
- Lambda expression support includes the following:
- **Functional interfaces**: An interface defining exactly one abstract method
  - Defining the available behavior for lambda expressions
- **Lambda expression syntax**: New syntax for writing lambda expressions
  - Can define new behavior (basically a method implementation)
  - **Without** requiring a complete class definition

Functional Interface

- A functional interface defines exactly one abstract method - as shown at bottom (1)
  - Very simple concept, but important for lambda expressions

- **`@FunctionalInterface`** may be used on a functional interface
  - Not required - only requirement is to define one abstract method
  - Using `@FunctionalInterface` on an interface that is not one, is an error (2)

```
@FunctionalInterface  // Optional, not used in Java library
public interface ActionListener extends EventListener {
  void actionPerformed(ActionEvent e);
}
```

Lambda Expressions

- A lambda expression can replace a class implementing a functional interface
  - It's a method definition with the containing class inferred
- Lambda expression syntax:

```
(parameters) -> body
```

  - **parameters**: Comma separated list of formal parameters
  - **->** : The arrow token
  - **body**: A single expression or statement block
  - The example below replaces our previous class definition (1)

```
JButton submitButton = new JButton("Press Me");
submitButton.addActionListener(
  (ActionEvent e) -> System.out.println("Button Pressed")
);
```

How Lambda Expressions Work

- The lambda expression is resolved using **type inference**
  - `button.addActionListener()` expects an `ActionListener`
  - So our lambda expression is of type `ActionListener`
  - It replaces an implementing class containing the method
  - The target type **MUST be a functional interface**

- The lambda expression must match the functional interface method
  - The code below will not compile
  - The expected `ActionEvent` parameter is missing

```
JButton submitButton = new JButton("Press Me");
submitButton.addActionListener(
  () -> System.out.println("Button Pressed")
);
```

Lambda Expression Syntax

- You can omit the data type of lambda expression parameters
  - You can also omit the parentheses if there is only 1 parameter
  - The following is also legal for our expression

```
e -> System.out.println("Button Pressed")
```

- You can use a block instead of a statement
  - You can also use the passed in parameters in your code
  - The example at bottom illustrates this

```
JButton submitButton = new JButton("Press Me");
submitButton.addActionListener( e-> {
  System.out.println("Button Pressed");
  System.out.println("Pressed at: " + new Date(e.getWhen()));
} );
```

Summary

- Lambda expressions are a powerful feature to write more concise, more readable code
  - Much less verbose than writing a complete class
    - Compare the class definition vs. the lambda expression
  - Defines the code in the same location as where it's used
    - Code is more understandable and maintainable
- There is much more functionality available
  - Covered in more advanced courses
- Like all powerful capabilities, be careful how you use them
  - Lambda expressions, while useful, have some pitfalls, including:
  - They are **not reusable** to the same extent as a class definition
  - The **syntax is somewhat awkward** and messy
  - Can be **hard to maintain** if you define a lot of code in them

Other Java Features

Assertions

Type-Safe Enums

Annotations

Lambda Expressions

Other Java Features

XML and Web Service Support

- The JDK contains all the XML and Web Service APIs
  - In earlier Java releases, these were part of Java EE only
- You can parse XML without any additional libraries
  - Using JAXP, SAX/StAX/DOM, JAXB, etc.
  - These APIs are all included
- You can write Web Services and Web Service clients
  - For example, a Web service client can be written and run without any other jars

Java DB

- The Apache Derby database is bundled with the JDK
  - It is called Java DB, and is just a redistribution of Derby
  - Production quality for small scale applications
  - Installed by default into *<java>\db* (1)

- Developers can be sure of having a database easily available
  - Samples or demo programs can use it

- It is NOT bundled with the JRE – only with the full JDK
  - They did not want to increase the size of the JRE with the Derby libraries
  - Some distributions (e.g. Java for Mac OS X) may not include it

Scripting Language Integration

- The JDK includes support for integrating with scripting languages
  - Any scripting language supporting JSR-223 (Scripting for the Java platform) can be integrated with the JDK
  - Java 6+ includes Mozilla Rhino for JavaScript support
  - Many scripting engines support JSR-223 – e.g. Python, Groovy, Ruby, JavaScript, AWK, …

- Can, for example
  - Invoke a Perl script from Java to do fancy string processing
  - Invoke on Java objects from JavaScript to do database access

Monitoring and Management Tools

- **JConsole** provides a visual interface to monitor:
  - Memory usage and garbage collection activities
  - Thread state, thread stack trace, and locks
  - Number of objects pending for finalization
  - Runtime information (e.g. process uptime and the CPU time)
  - VM information (e.g. JVM input arguments and class path)
- **jstat**: Provides VM statistics
  - Including memory usage, garbage collection time, class loading, and the just-in-time compiler statistics
- **jmap/jhat**: Provides heap dump analysis
- **jstack**: Provides thread stack trace

Other Features

- **Java 7 and earlier**
  - **Integration with Security services** like PKI, and Kerberos, new XML Digital Signature support, and more
  - **Pluggable Annotations**: Simplifies defining/processing annotations
  - **Better support for dynamically typed languages** (JSR 292)
    - More extensive scripting support, much better performance than JSR 223
  - **I/O improvements and additions** (NIO.2)
  - **Upgraded XML and Web Services stacks**
- **Java 8**
  - **JDBC upgrade** to 4.2
  - **Lambda expression integration** with I/O, collections, other libraries
  - **Annotation improvements**: Annotations anywhere a type is used
  - **Security** enhancements
  - **New Date/Time** package

Session 14: I/O Streams (Optional)

`Readers` and `Writers`
Byte Streams
Formatted Output
New I/O (NIO) APIs

Session Objectives

- Describe Java I/O with character streams and the `Reader` and `Writer` classes
- Explain high-level and low-level streams and their interactions
- Outline the Java byte streams, the `InputStream` and `OutputStream` classes and conversion between byte streams and character streams
- Demonstrate formatted output with Java streams

Readers and Writers

Readers and Writers
Byte Streams
Formatted Output
New I/O (NIO) APIs

Overview of I/O Streams

- The `java.io` classes abstract I/O operations
  - Done in a hardware-independent manner
  - Sources and destinations can be anything that can contain data
  - Use the same interface for different devices
    - A network socket, a file, an in-memory array
  - The data can be of any type: objects, characters, images, etc.

- I/O can be done with bytes or Unicode characters
  - `InputStreams` and `OutputStreams` read and write all data (in the form of bytes)
  - `Readers` and `Writers` work with (Unicode) character data

Character Streams

- Character streams operate on 16-bit Unicode characters
  - Byte streams only support I/O on bytes
  - The character streams were added after the byte-based streams
- Use `Reader` and `Writer` classes and their subclasses
- Provide support for different character encodings
  - Automatically translate from Unicode to local character set
  - Make applications easy to internationalize
  - Reading files in different formats is easy and transparent
  - Can convert between bytes and characters, according to a specified encoding

Class Reader

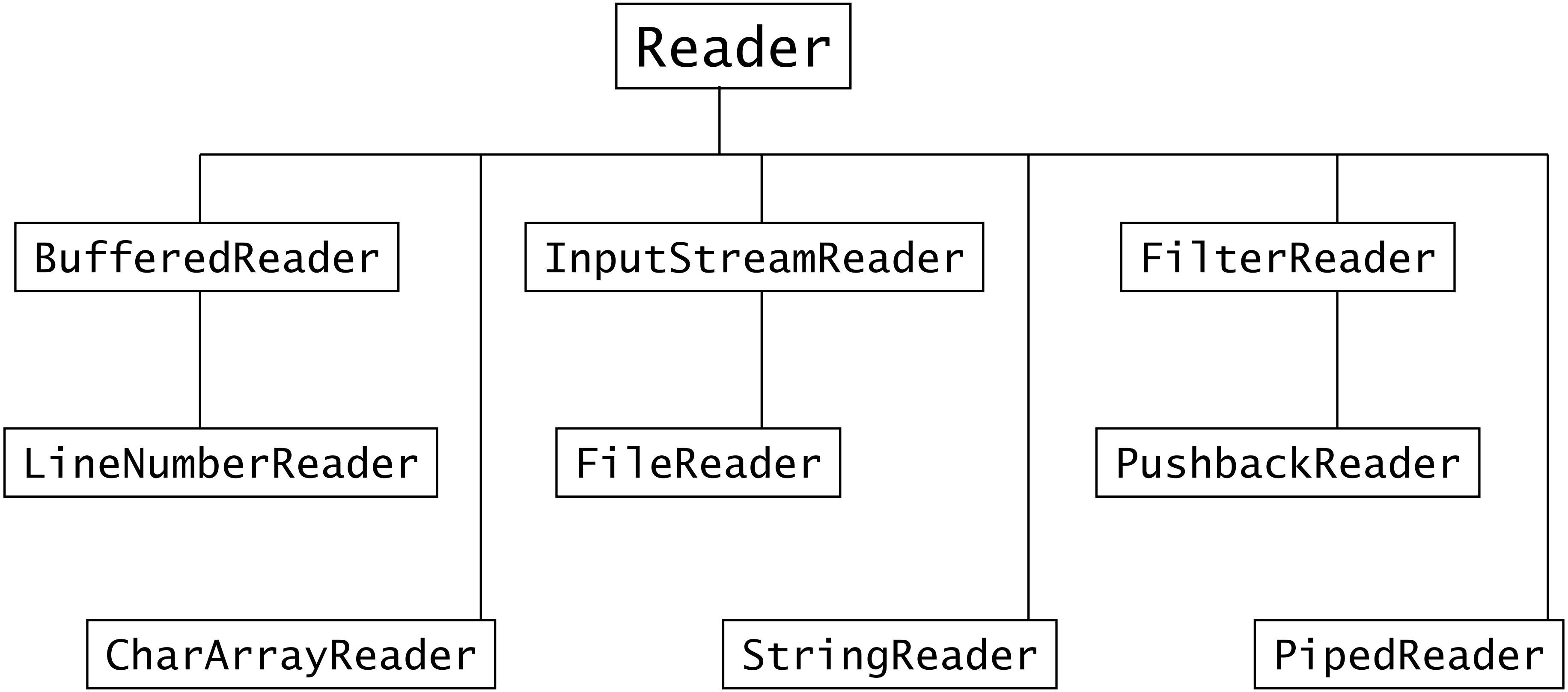
- **Reader**: Abstract base class for character input – public methods include:
  - `int` **`read`**`()`: Read a single character
  - `int` **`read`**`(char[] cbuf)`: Read characters into an array
  - `abstract int` **`read`**`(byte[] cbuf,int off,int len)`: Reads characters into a portion of an array
  - `abstract void` **`close`**`()`: Close the stream
  - `boolean` **`ready`**`()`: Is this stream ready to be read?
  - `int` **`skip`**`(long n)`: Skip n characters
- Most stream methods can throw an **`IOException`**
- Read methods **block** if input isn't available to be read

Class `Writer`

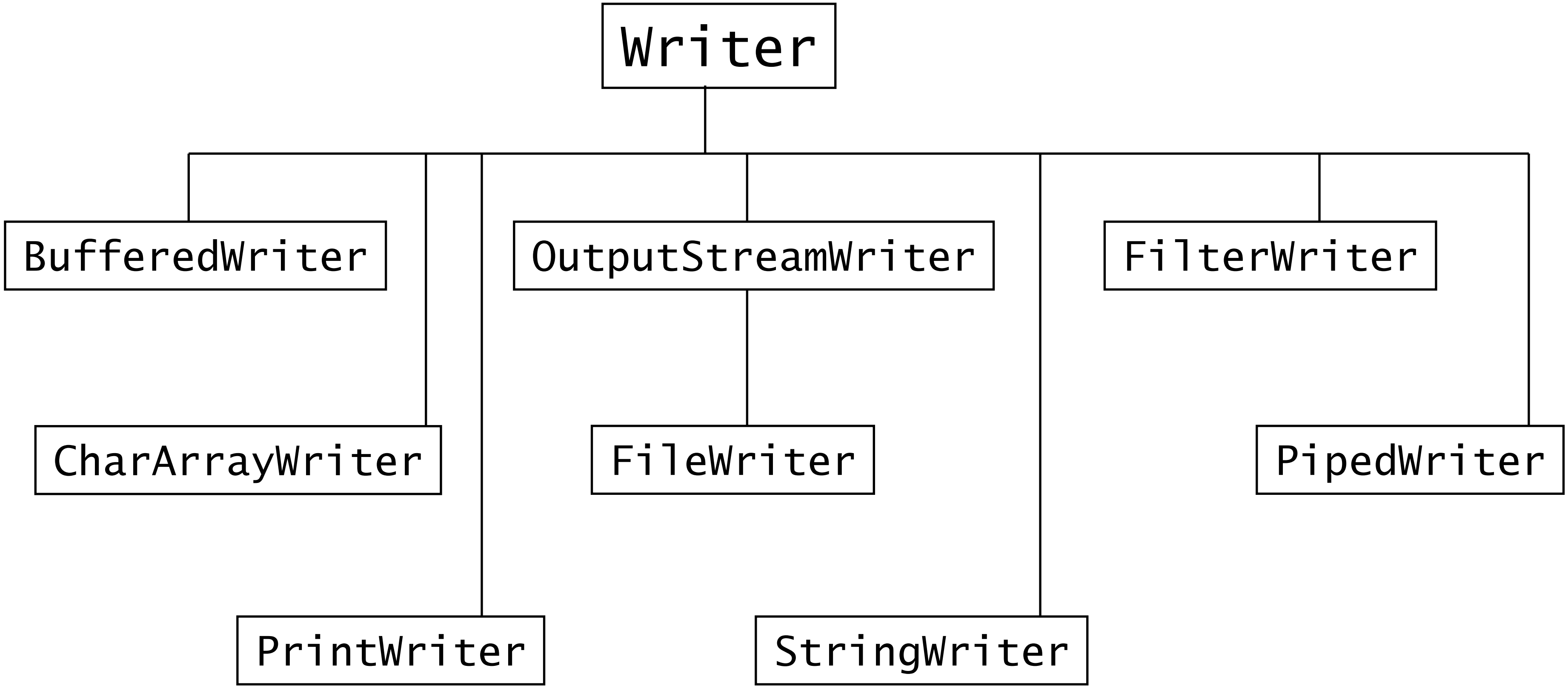
- `Writer`: Abstract base class for character output – public methods include:
  - `void write(int c)`: Write one character
  - `void write(char[] cbuf)`: Write an array of characters
  - `abstract void write(char[] cbuf,int off,int len)`: Write a portion of an array of characters
  - `void write(String str)` – writes a `String`
  - `void write(String str,int off,int len)` – writes a portion of a `String`
  - `abstract void flush()` – flushes the stream
  - `abstract void close()` – closes the stream, **flushing it first**

- All of these methods can throw an `IOException`

Common Reader Subclasses



Common Writer Subclasses



Using Readers and Writers

- `FileReader`: Concrete subclass of `Reader` for file reading
- `FileWriter`: Concrete subclass of `Writer`, for file writing
- They use the default character encoding to read/write files
  - They'll work with your ASCII text files, and convert to and from 16-bit characters as needed
  - They put the unsigned value of each character into an `int`, and return `-1` on EOF

Using Readers and Writers - Example

```
import java.io.*;
// read from foo.txt and write to bar.txt
try (  // try-with-resources works here
  FileReader fr = new FileReader("foo.txt");
  FileWriter fw = new FileWriter("bar.txt");
) {
  int c;
  // read until EndOfFile: indicated by -1
  while ((c=fr.read()) != -1) {
    fw.write(c);
  }
} // No need to close Reader/Writer with try-with-resources
catch (FileNotFoundException e) {
  System.out.println(e);
}
catch (IOException e) {
  System.out.println(e);
}
```

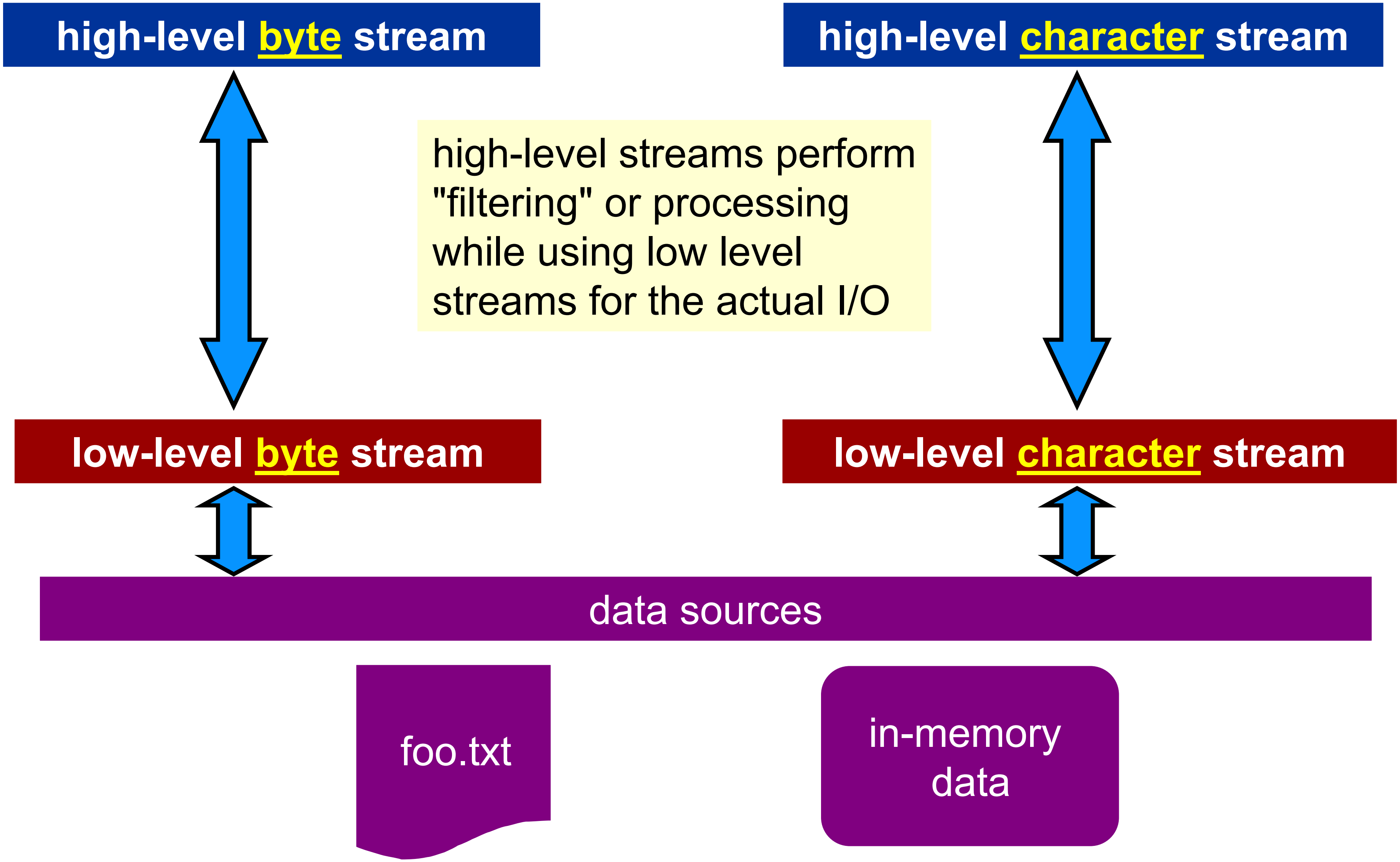
Path Separators

- `class File` defines constants for system-dependent separators
  - Supports **platform independent** filenames and lists

  - `static final char` **`pathSeparatorChar`**: Path separator character
  - `static final String` **`pathSeparator`**: Path separator character, as a `String`
    - Separates sequences of files or directories, as a "path list"
      - **`;`** on Windows, **`:`** on Unix

  - `static final char` **`separatorChar`**: File separator character
  - `static final String` **`separator`**: File separator character, as a `String`
    - Separates the directory and file components of a filename
      - **** on Windows, **/** on Unix

High-Level (Filtering) Streams



BufferedReader - Filtering Stream Example

- `FileReader` reads character-by-character
  - We can wrap a `BufferedReader` around a `FileReader`
  - It can read a file line-by-line,
  - Below, we read the input line-by-line and write it to standard output

```
// copy text from foo.txt to standard out
// wrap a BufferedReader around a FileReader
try ( BufferedReader br = new BufferedReader(
                                new FileReader("foo.txt")); ) {
  String curLine;                // temp storage for each line
  // read until EndOfFile: indicated by null
  while ((curLine=br.readLine()) != null)
  {
    System.out.println(curLine);
  }
}                                // catch blocks omitted
```

Byte Streams

Readers and Writers

Byte Streams

Formatted Output

New I/O (NIO) APIs

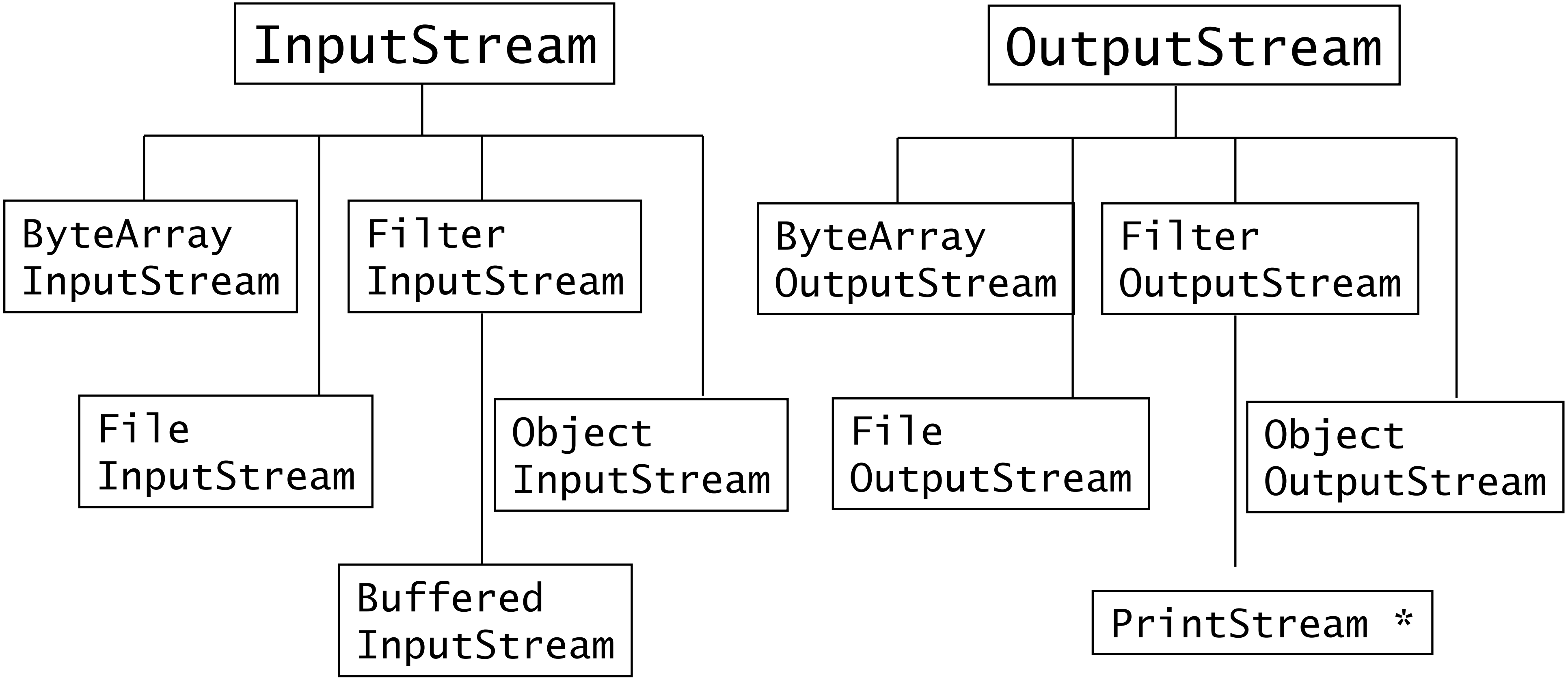
Byte Stream Classes

- `java.io` includes a number of byte streams
  - Some have generally been superceded by the `Reader`/`Writer`
    - That do character I/O
  - Others are still useful for I/O of other types

- **`InputStream`** / **`OutputStream`**: Abstract base classes defining functionality of all byte-based streams
  - Analogous to classes `Reader` and `Writer`
  - Their methods are similar – see the Javadoc

- There are also `FilterInputStreams` and `FilterOutputStreams`

Common Stream Subclasses



Converting Between Byte & Character Streams

- Two "bridge" streams convert between byte and character streams
- **InputStreamReader**: Concrete `Reader` subclass taking an `InputStream` in its constructor
  - Feed the `InputStreamReader` to another `Reader` (e.g. BufferedReader) to bridge the `InputStream` to it
  - It **translates bytes into characters**, according to a specified encoding

```
InputStreamReader bridges InputStream to Reader
                           (bytes) -> (characters)

BufferedReader br = new BufferedReader(
  new InputStreamReader(System.in));
```

Converting Between Byte & Character Streams

- **`OutputStreamWriter`**: Concrete `Writer` subclass taking an `OutputStream` in its constructor
  - **Translates characters into bytes**, according to a specified encoding

```
OutputStreamWriter bridges Writer to OutputStream
                        (characters) -> (bytes)

BufferedWriter bw = new BufferedWriter(
  new OutputStreamWriter(System.out));
```

Character Stream & Byte Stream Equivalents

Character stream class	Description	Corresponding byte stream class
Reader	Abstract class for character input streams	InputStream
BufferedReader	Buffers input, parses lines	BufferedInputStream
InputStreamReader	Translates a byte stream into a character stream	(none)
FileReader	Translates bytes from a file into a character stream	FileInputStream
FilterReader	Abstract class for filtered character input	FilterInputStream
Writer	Abstract class for character output streams	OutputStream
BufferedWriter	Buffers output, uses platform's line separator	BufferedOutputStream
FilterWriter	Abstract class for filtered character output	FilterOutputStream
OutputStreamWriter	Translates a character stream into a byte stream	(none)
FileWriter	Translates a character stream into a byte file	FileOutputStream
PrintWriter	Prints values and objects to a Writer	PrintStream
StringWriter	Writes to a String	(none)

Formatted Output

Readers and Writers

Byte Streams

Formatted Output

New I/O (NIO) APIs

Formatted Output

- PrintStream provides methods for formatted output:

```
PrintStream printf(String format, Object... args)
PrintStream format(String format,  Object... args)
```

  - The two methods work identically [(1)]
  - The format string can include embedded format specifiers
    - They're replaced by formatted data using the values in args
    - The rest of the format string is output unchanged

- Below, **%s** specifies "format as string"
  - There are two occurrences of %s in the format string
  - They are replaced by the two data objects passed in

```
// The output is: My full name is Alan Turing
System.out.printf("My full name is %s %s",
                  "Alan", "Turing");
```

Integer Format Specifiers

- Applicable to `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, `Long`, and `BigInteger`
  - **`'d'`**: Format output as decimal integer
  - **`'o'`**: Format output as base eight integer
  - **`'x'`**: Format output as hex integer
  - **`'X'`**: Format output as hex integer using upper case letters
- The example below
  - Includes a newline (\n) in the first output string
  - Shows a decimal value usage printing: **`I am 104 years old`**

```
System.out.printf("My full name is %s %s\n",
                  "Alan", "Turing");
System.out.printf("I am %d years old", 104);
```

Format Specifier Modifiers

- Format specifiers can include optional modifiers with the form:

  `%[argument_index$][flags][width][.precision]conversion`

  –**`argument_index`**: Decimal integer indicating the position of the argument in the argument list

  - The first argument is referenced by "1$", the second by "2$", etc.

  –**`flags`**: Set of characters that modify the output format

  –**`width`**: Non-negative decimal integer indicating the minimum number of characters to be written to the output.

  –**`precision`**: Non-negative decimal integer usually used to restrict the number of characters

  –**`conversion`**: (e.g. s) Indicates how the argument should be formatted (e.g. to a string)

  - If there are lower and upper case versions (e.g. %x and %X) the upper case version converts results to upper case

Format Specifier Modifiers Example

- Below, we specify a minimum width of 15 and show its output
- At bottom, we specify (via the "," flag) that locale-specific group separators be used
- Many, many flags - different for each conversion
  - See the `java.util.Formatter` javadoc for extensive details

```
System.out.printf("My full name is %15s %15s\n",
                  "Alan", "Turing");
System.out.printf("My full name is %15s %15s\n",
                  "Viswanathan", "Anand");
```

```
My full name is            Alan          Turing
My full name is     Viswanathan           Anand
```

```
System.out.printf("Dinosaurs disappeared %,d years ago",
                  65000000);
```

```
Dinosaurs disappeared 65,000,000 years ago
```

Other Format Specifiers

- **Float**: Applicable to `float, Float, double and Double`
  - `'e'`: Format output using computerized scientific notation.
  - `'g'`: Format output in general scientific notation
  - `'f'`: Format output using decimal format.
  - `'a'`: Format output in hexadecimal exponential form.
- **Date/Time**: Large number of specifiers
  - Consist of prefix %t (or %T), followed by various suffixes, e.g.
  - We list a few here, with output for Nov. 14, 2013 at 12:01:13 PM
  - `'tI'`: Hour of day for 12 hr. clock (and many others for hour, minutes, seconds, etc) - `12`
  - `'tr'`: Complete time in 12 hour clock - `12:01:13 PM`
  - `'tB'`: Complete month name - `November`
  - `'tD'`: Complete date as m/d/y - `11/15/13`

Summary

- Java supports flexible output formatting
  - Using familiar format specifiers common in industry

- Formatting can be used to create strings
  - Via the following `String` method

```
static String format(String format, Object... args)
```

  - Returns a formatted string using the specified format string and arguments

- All format methods have a version accepting a locale, e.g.

```
PrintStream printf(Locale locale, String format,
  Object... args)
```

  - Formats based on the specified locale

New I/O (NIO) APIs

Readers and Writers

Byte Streams

Formatted Output

New I/O (NIO) APIs

New I/O (NIO and NIO.2)

- I/O facilities introduced in Java 1.4 (NIO) and Java 7 (NIO.2) provided new features and improved performance in:
  - Buffer management
  - Scalable network and file I/O
  - Character set support
  - Regular expression matching
  - File access

- The root package is **`java.nio`**, with several subpackages
  - NIO does not replace `java.io`, but rather supplements it

NIO Features

- `java.nio` provides sophisticated buffers for primitive types
  – Used throughout the rest of the NIO APIs
  – Byte buffers can map a region of a file directly into memory for memory-resident file I/O

- `java.nio.channels` introduces ***channels***
  – Connections to entities that can perform I/O, e.g., a hardware device, file, or socket
  – They offer multiplexed, non-blocking I/O

- `java.nio.charset` provides classes that can perform character set encoding/decoding
  – Very fast and efficient, leveraging buffers and channels

NIO Features

- `java.util.regex` offers regular expressions
  - Which we looked at briefly in Session 6
  - A regular expression can be used just once or repeatedly
- `String` offers a convenient `matches` method for regular expressions that will be used just once
- A regular expression can also be compiled into a `Pattern` object
  - Useful when you need to evaluate data against a regular expression multiple times

[Optional] Lab 14.1: Formatted Output

In this lab, you will use the Java formatting capabilities

Review Questions

1. What is the difference between character stream I/O and byte stream I/O?
2. True of false: the Java byte streams are no longer useful, but kept only for compatibility.
3. What is meant by filtering?
4. What formatting capabilities are available in Java I/O streams

Lesson Summary

- **Byte stream** I/O supports I/O on bytes
  - Byte streams work will with non-character data, but are difficult to use with Unicode character I/O
- **Character streams** do character-based I/O
  - They support the Unicode character set
  - Including support for different encodings, and I18N
- **Filter streams** wrap around a core I/O stream to process the data in some way
  - e.g. `BufferedInputStream` adds buffering for efficiency
- Java streams support **formatting** capabilities
  - To easily mix textual output with formatting instructions
  - Much easier than string concatenation

Recap

What We've Learned

- Covered important areas for writing good OO code in Java:
  - Writing and using well structured **Java classes**
  - Using **packages** and **composition** to handle complex types and systems
  - Using **inheritance**, **polymorphism**, and **interfaces** to create type hierarchies and decrease coupling
  - Handling errors with **exceptions**
  - Accessing relational databases with **JDBC**
  - Learning about Java libraries like **Collections** and **I/O**
- There is still a lot of opportunity for learning
  - In particular, object modeling and diagramming are important tools for building OO systems
  - This is a good area for your next steps in learning

Resources

Web Resources

- **Java Home Page**:
  - http://www.oracle.com/technetwork/java/index.html
- The **Java Tutorial**: http://docs.oracle.com/javase/tutorial/
- **Java Ranch** (good beginner site): http://www.javaranch.com/

Books

- **Core Java** by Cay Horstmann and Gary Cornell
- **UML Distilled** by Martin Fowler (A brief guide to the Unified Modeling Language)
- **Applying UML and Patterns** by Craig Larman
  - Introduction to OO Analysis and Design
- **Head First Design Patterns** by Elisabeth Freeman et al.
  - Introduction to OO software design using design patterns

Appendix: JDBC
Java Database Connectivity

JDBC Overview

JDBC Architecture and API

Database Connections

Issuing Statements and Processing Data

Session Objectives

- Explain JDBC's role in database access
- Describe the JDBC architecture and the roles of the JDBC driver and database URL
- Describe the Factory Pattern and explain why it's used in the JDBC API
- Connect to a database via JDBC
- Execute a statement against the database and process the returned rows

JDBC Overview

JDBC Overview

JDBC Architecture and API

Database Connections

Issuing Statements and Processing Data

Relational Database Description

- Access to a database is a crucial part of most multiuser applications

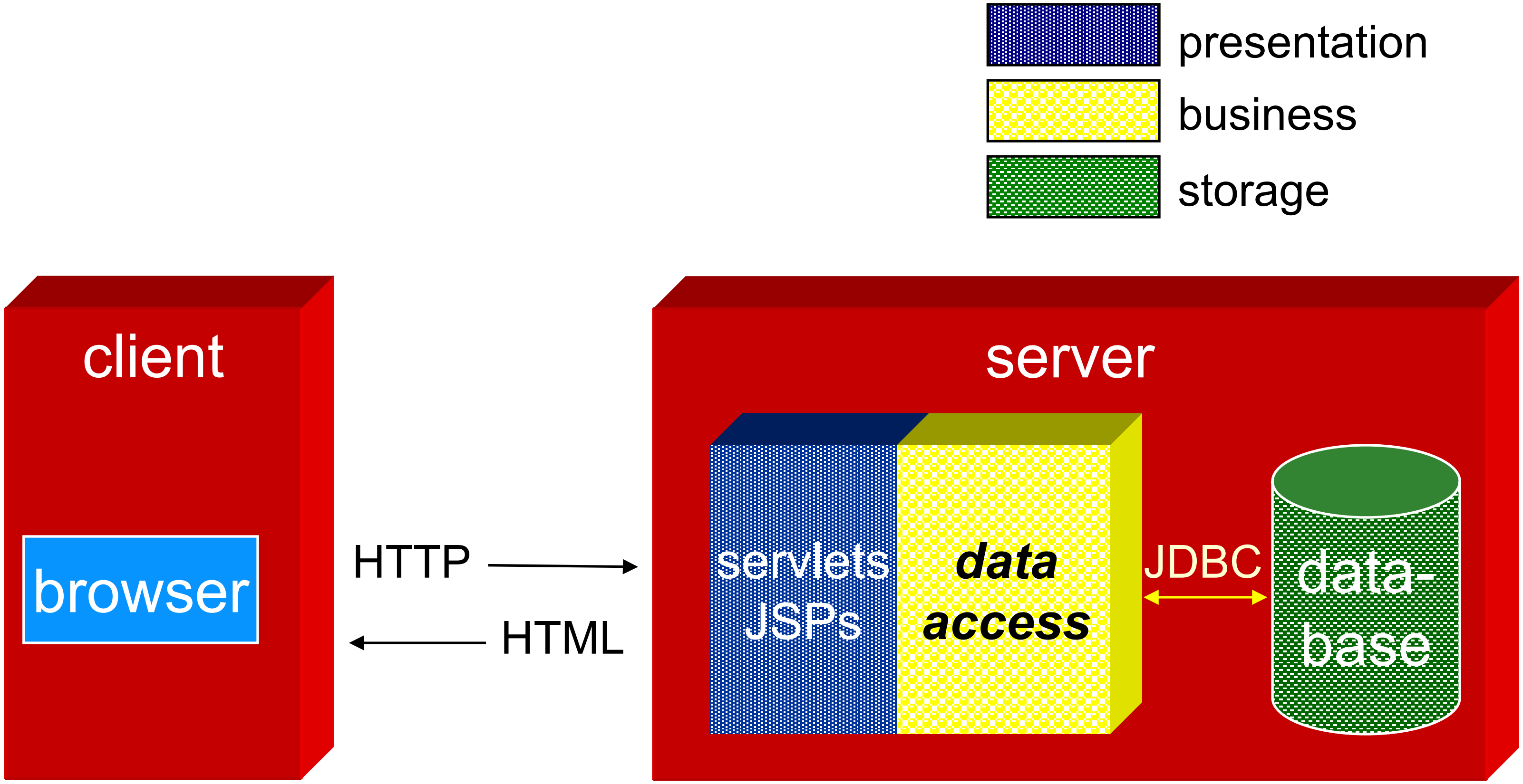
- ***Relational databases*** use tables to store data
  - A table consists of rows and columns
  - Each column has a **name** and a **type** – character, numeric, date, etc.
  - Each row represents an ***entity*** – a collection of related data
  - The intersection of a row and a column is called a ***field***

- For example, here's an **`Employee`** table:

ID	Name	Salary	Hiredate
1	Jacob	1606	1995-01-02
2	Ted	NULL	1995-01-02
3	Laurentino	1388	1995-01-02

Web-Based Data Access Architecture

- This is a popular architecture for Web applications



What is JDBC?

- **JDBC = Java Database Connectivity**
  – Standard API included with the JDK

- Generic SQL database access framework for Java

- Based on X/Open CLI (Call-Level Interface)
  – The basis for ODBC (Open Database Connectivity, a Microsoft standard)

- Included in the Java core API libraries as of Java 1.1

JDBC Characteristics

- Call-level SQL interface for Java
  - Executes raw SQL statements
  - Retrieves the results
  - Modeled after ODBC
- Completely defined in the Java language
- Provides a uniform interface
  - To different databases
  - Under different connectivity situations
- Leverages separation of interface from implementation
  - Standard API with pluggable vendor implementations (***drivers***)

JDBC Specification and Packages

- **JDBC 4.2 is the specification included in Java 8**
  - Consolidates all previous JDBC specifications (see notes)

- Consists of two packages:
  - **`java.sql`** the core JDBC API
  - **`javax.sql`** Enterprise capabilities (e.g. DataSource)

  - Both packages are now part of the Java JDK

JDBC Architecture and API

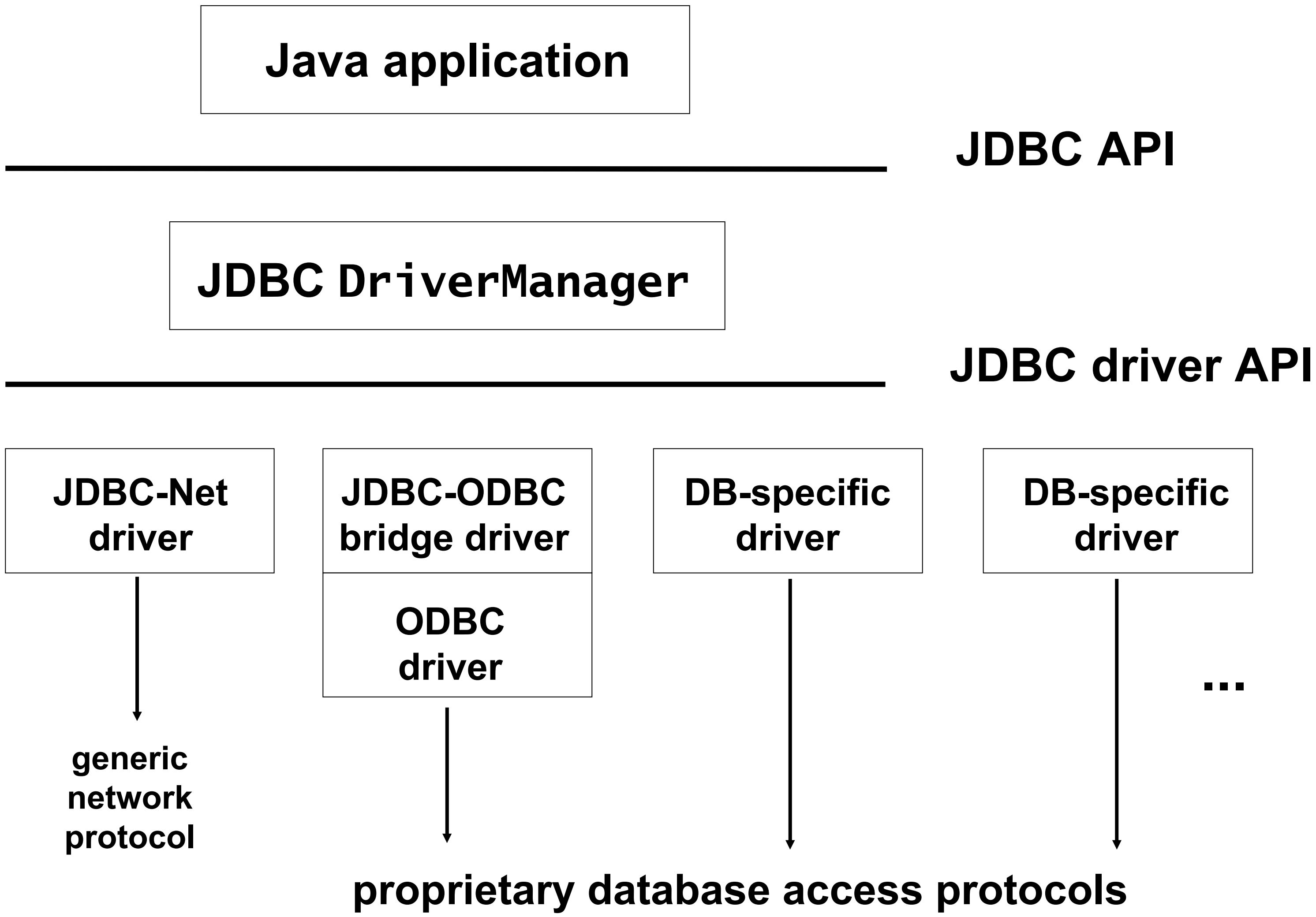
JDBC Overview

JDBC Architecture and API

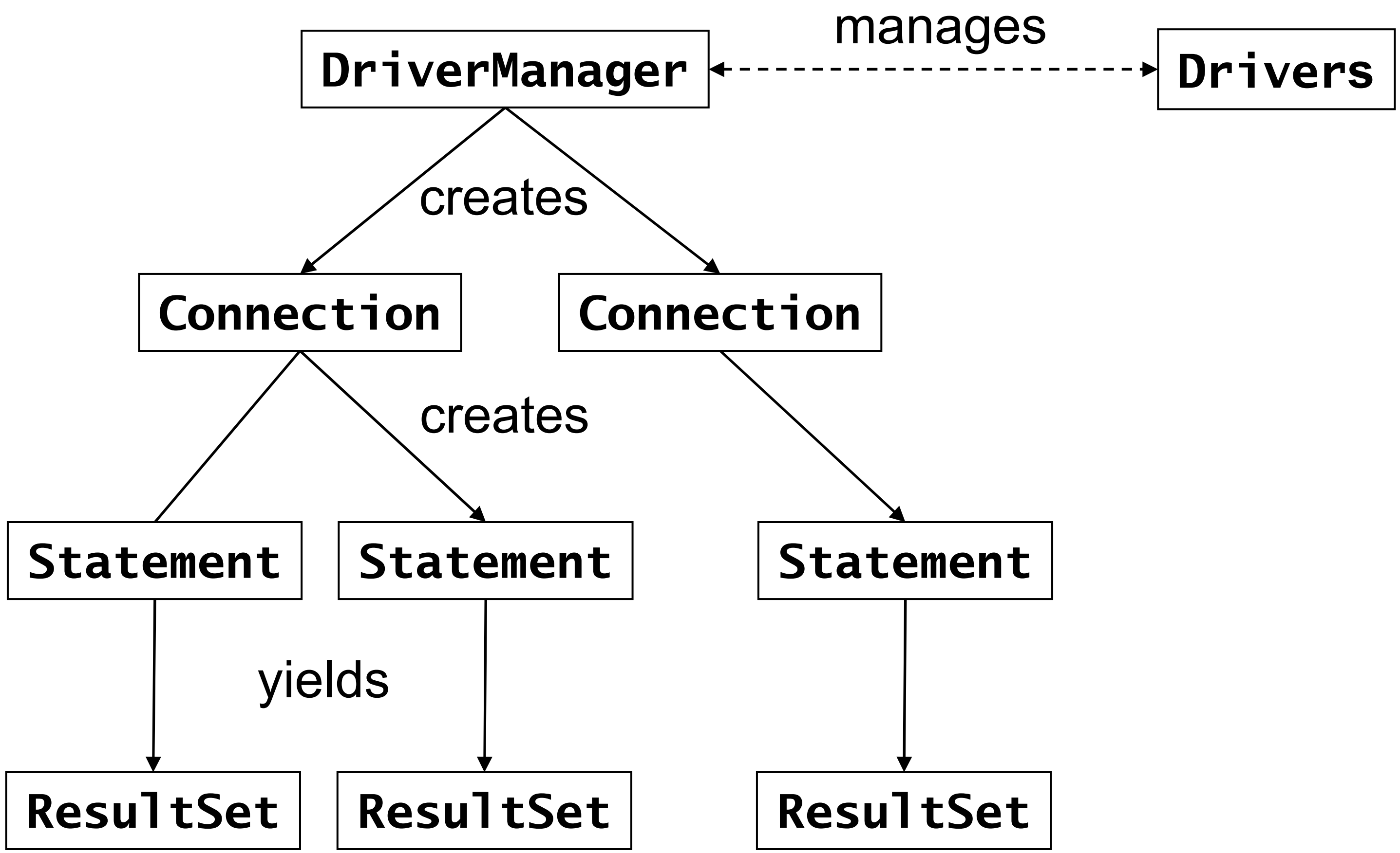
Database Connections

Issuing Statements and Processing Data

JDBC Architecture



The Fundamental JDBC API



JDBC instantiation diagram (Factory Pattern)

The `DriverManager` Class

- Management layer between application and database drivers

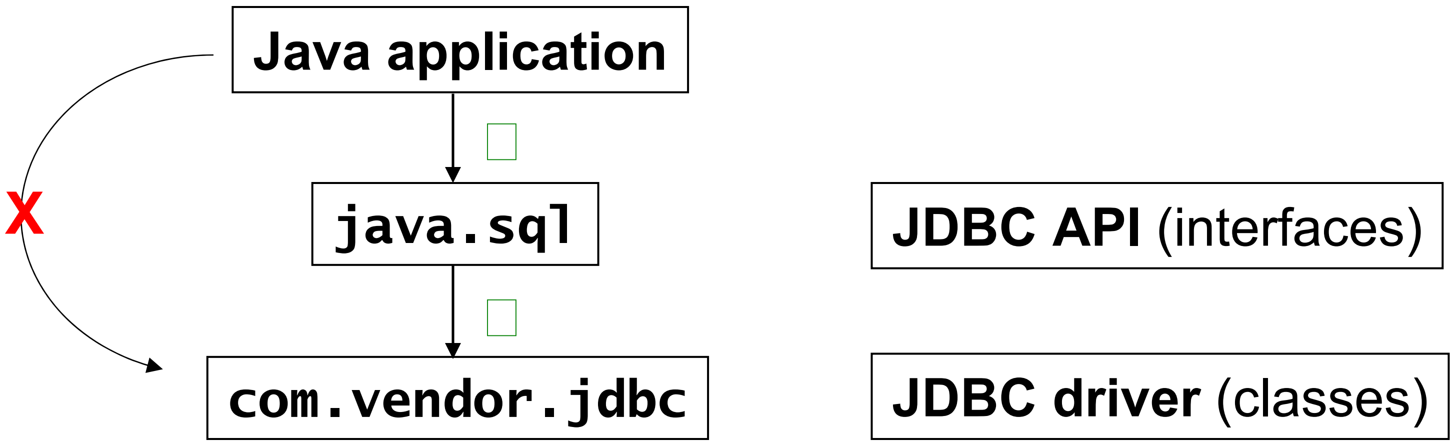
- Keeps track of drivers
  - `DriverManager` will automatically load a driver when asked for a connection
  - Note that from JDBC 4.0 on (Java 6) it is not necessary to load a driver manually in your code (1)

- Manages connections between drivers and databases

- Keeps track of global state
  - Security, login, timeout, logging

What is a JDBC Driver?

- Set of **classes** implementing the **interfaces** defined in JDBC
- The driver classes are not included in the JDBC API
  - The JDBC interface definitions are, but the drivers themselves are supplied by third parties, generally the database vendors
  - They are usually packaged into JAR files
  - They must be on the classpath at runtime



Naming Databases with URLs

- A URL (Universal Resource Locator) is a general way to locate a resource
  - Many are familiar with URLs based on HTTP, e.g., *http://www.LearningPatterns.com*
- A JDBC URL:
  - Provides a way of identifying a database so that an appropriate driver will recognize it
  - Provides the information needed for that driver to establish a connection with that database

JDBC URL Syntax

- **`jdbc:<subprotocol>:<subname>`**
  - Protocol is always **`jdbc`**
- `<subprotocol>` - specifies how you connect to the database
  - For example, **`oracle`**, **`db2`**, **`sybase`**, **`derby`**, **`odbc`**
- `<subname>` - identifies a particular database
  - For example, the **`Bank`** database on host machine **`venus`**
- Complete example:
  - `jdbc:`**`vendor`**`://`**`venus`**`/`**`Bank`**
- Sample URLs for various databases
  - **Oracle** `jdbc:`**`oracle`**`:thin:@`**`venus`**`:`**`Bank`**
  - **IBM DB2** `jdbc:`**`db2`**`://`**`venus`**`/`**`Bank`**
  - **Derby** `jdbc:`**`derby`**`://`**`venus:1527`**`/`**`Bank`**
  - **JDBC-ODBC** bridge `jdbc:`**`odbc`**`:`**`Bank`**

LEARNINGPATTERNS

Database Connections

JDBC Overview

JDBC Architecture and API

Database Connections

Issuing Statements and Processing Data

Database Connections

- A JDBC `Connection` object represents a session with a specific database
- SQL statements are executed and results are returned via a `Connection`
- An application can open multiple connections to a single database or to multiple databases
- The Connection interface contains
  - Methods to create statements
  - Methods to get "metadata" about the connection (its capabilities)
  - Methods to manage the connection
    - Read-only, Transaction mode (isolation level), Auto-commit behavior, Close
  - Methods to manage transactions
    - Commit, rollback, savepoints

Establishing a Database Connection

- You get a `Connection` object via one of the `static` **`DriverManager.getConnection`** methods
  - The `Connection` object returned is open (connected)

```
static Connection getConnection(String url)
throws SQLException

static Connection getConnection(
  String url, String user, String password)
throws SQLException

static Connection getConnection(String url, Properties info)
throws SQLException
```

Using try-with-resources

- Below, we use try-with-resources with getting a JDBC connection
  - Very easy and clean - available in Java 7+

```
// Load driver via Class.forName() here *

// get a connection to the database using try-with-resource
try (Connection conn =   DriverManager.getConnection(
  "jdbc:derby://localhost:1527/PERSONNEL",
  "guest", "password"); ) {

  // rest of code
}
catch (SQLException sqle) {
  System.out.println(sqle);
}
// No finally or close of connection needed
```

Closing the Connection

- **It is imperative that you close your connection when you are done with it**
  - If your application terminates without doing so, it may leave a "ghost" or "dangling" connection on the server side
  - Using `try-with-resources` does this automatically

- Every client-side JDBC object has a server-side counterpart implemented by the DBMS
  - Your JDBC `Connection` object is a thread or a process or something on the server side
  - Closing your JDBC objects ensures that these server-side resources get freed up

Issuing Statements and Processing Data

JDBC Overview

JDBC Architecture and API

Database Connections

Issuing Statements and Processing Data

Creating Statements

- You get various JDBC statement objects from a `Connection`
- **`createStatement`** creates a **`Statement`**
  - Used for simple and dynamic SQL statements
- **`prepareStatement`** creates a **`PreparedStatement`**
  - Used for "precompiled" SQL statements
  - They take `IN` parameters – more efficient than `Statements`
  - `PreparedStatement extends Statement`
- **`prepareCall`** creates a **`CallableStatement`**
  - Used for calling DBMS stored procedures
  - `CallableStatement extends PreparedStatement`

The Statement Interface

- Used to execute simple and dynamic SQL statements
  - Use `Statement` when the SQL is not known in advance, i.e., you do not know the table name, the column names, the nature of the statement (query or update), etc.
  - These are called ***dynamic*** SQL statements
- You get **`Statement`** objects from a `Connection`

```
Statement createStatement() throws SQLException
```

- The following methods are used to execute SQL with it:

```
ResultSet executeQuery(String sql)  throws SQLException
int       executeUpdate(String sql) throws SQLException

// not normally used (see notes)
boolean   execute(String sql)       throws SQLException
```

Using Statement - Example

```
// build the SQL
String sql = "SELECT ID, Name, Salary " +
  "FROM Employee WHERE Salary >= 1250";
try {
  // conn is an open Connection object as created earlier

  // See notes for using try-with-resources here
  // create a Statement object
  Statement stmt = conn.createStatement();

  // execute the SQL with it
  ResultSet rs = stmt.executeQuery(sql);

  // we'll continue after we discuss ResultSets
}
catch (SQLException sqle)
{
  System.out.println(sqle);
}
```

The ResultSet Interface

- Represents a tabular set of data generated by a `Statement`
  - It contains the column names and the data
- It maintains a ***cursor*** that points to the current row in the set
  - The cursor is initially positioned **before** the first row
  - Calling the **`next`** method advances the cursor by one row
  - `next` returns `true` if on a valid row, `false` if no more rows

```
boolean next() throws SQLException
```

- Here is the `ResultSet` of our Employee table query

cursor →

ID	Name	Salary
1	Jacob	1606
3	Laurentino	1388

Extracting Data from a ResultSet

- You use **get*XXX*** methods to retrieve the column values for the current row
  - `getInt`, `getDate`, `getString`, etc.
  - They map between SQL data types and Java data types, e.g.,
    - SQL VARCHAR to Java `String`
    - SQL INTEGER to Java `int`
  - Several conversions may work for a given type
- The column desired can be specified by its name or number
  - Column numbering starts at 1

```
XXX getXXX(String columnName) throws SQLException
XXX getXXX(int columnIndex)   throws SQLException
```

SQL -> Java Type Mappings

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte()	X	x	x	x	x	x	x	x	x	x	x	x	x												
getShort()	x	X	x	x	x	x	x	x	x	x	x	x	x												
getInt()	x	x	X	x	x	x	x	x	x	x	x	x	x												
getLong()	x	x	x	X	x	x	x	x	x	x	x	x	x												
getFloat()	x	x	x	x	X	x	x	x	x	x	x	x	x												
getDouble()	x	x	x	x	x	X	X	x	x	x	x	x	x												
getBigDecimal()	x	x	x	x	x	x	x	X	X	x	x	x	x												
getBoolean()	x	x	x	x	x	x	x	x	x	X	x	x	x												
getString()	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x						
getBytes()														X	X	x									
getDate()											x	x	x				X		x						
getTime()											x	x	x					X	x						
getTimeStamp()											x	x	x				x	x	X						
getAsciiStream()											x	x	X	x	x	x									
getUnicodeStream()											x	x	X	x	x	x									
getBinaryStream()														x	x	X									
getClob()																				X					
getBlob()																					X				
getArray()																						X			
getRef()																							X		
getCharacterStream()											x	x	X	x	x	x									
getObject()	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X

Extracting Data from a ResultSet - Example

```
// picking up where we left off

try
{
  // extract the data from the ResultSet
  while (rs.next())
  {
    int        id   = rs.getInt("ID");
    String     name = rs.getString("Name");
    BigDecimal sal  = rs.getBigDecimal("Salary");

    // do something with the data - create a value object
    EmployeeValue emp = new EmployeeValue(id, name, sal);
  }
}
catch (SQLException sqle)
{
  System.out.println(sqle);
}
```

Prepared Statements

- SQL that is to be executed multiple times can be "precompiled" and stored in a `PreparedStatement` object
  - The precompiled SQL is "parameterized" with **?** placeholders for the literal values in the SQL statement

```
SELECT ID, Name, Salary
FROM Employee WHERE Salary >= ?
```

- You use **set*XXX*** methods to set the **?** IN parameters

```
void setXXX(int paramIndex, XXX value) throws SQLException
```

- The following methods are commonly used to execute it:
  - Note that there is no SQL passed into the execute methods

```
ResultSet executeQuery()  throws SQLException
int       executeUpdate() throws SQLException
```

Using PreparedStatement - Example

```
// define the ?-SQL
String sql = "SELECT ID, Name, Salary " +
  "FROM Employee WHERE Salary >= ?";

try {
  // conn is an open Connection object as created earlier

  // prepare the ?-SQL with the DBMS PreparedStatement
  PreparedStatement pstmt = conn.prepareStatement(sql);

  // set the ? - they are numbered starting at 1
  pstmt.setBigDecimal(1, new BigDecimal("100.00"));

  // execute the PreparedStatement and get a ResultSet
  ResultSet rs = pstmt.executeQuery();
}
catch (SQLException sqle) {
  System.out.println(sqle);
}
```