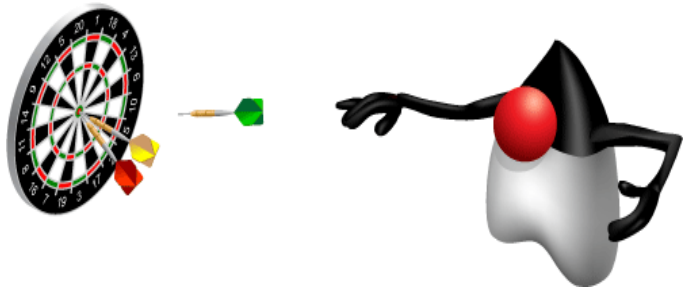
Java Syntax and Class Review

Objectives

After completing this lesson, you should be able to do the following:

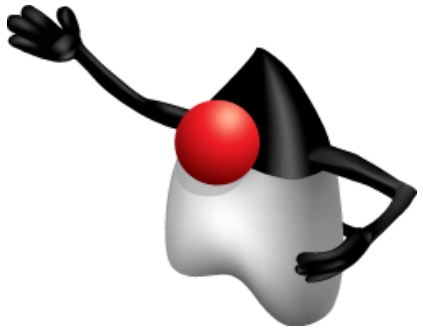
- Create simple Java classes
  - Create primitive variables
  - Use operators
  - Create and manipulate strings
  - Manage Flow Control:
    - Use `if-else` and `switch` statements
    - Iterate with loops: `while`, `do-while`, `for`, enhanced `for`
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements



Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Basic `if-else` and `switch` branching constructs
- Iteration with `for` and `while` loops



Java Class Structure

```
package <package_name>;

import <other_packages>;

public class ClassName {
   <variables(also known as fields)>;

   <constructor(s)>;

   <other methods>;
}
```

A Simple Class

A simple Java class with a `main` method:

```
public class Simple {

    public static void main(String args[]){

    }
}
```

Java Naming Conventions

```
public class CreditCard {
    public final int VISA = 5001;
    public String accountName;
    public String cardNumber;
    public Date expDate;

    public double getCharges(){
        // ...
    }

    public void disputeCharge(String chargeId, float amount){
        // ...
    }
}
```

Class names are nouns in upper camel case.

Constants should be declared in all uppercase. letters

Variable names are short but meaningful in lower camel case.

Methods should be verbs, in lower camel case.

How to Compile and Run

Java class files must be compiled before running them.
To compile a Java source file, use the Java compiler (`javac`).

```
javac -cp <path to other classes> -d <compiler output
path> <path to source>.java
```

- You can use the `CLASSPATH` environment variable to the directory above the location of the package hierarchy.
- After compiling the source `.java` file, a `.class` file is generated.
- To run the Java application, run it using the Java interpreter (`java`):

```
java -cp <path to other classes> <package
name>.<classname>
```

How to Compile and Run: Example

- Assume that the class shown in the notes is in the directory `test` in the path `/home/oracle`:

```
$ javac HelloWorld.java
```

- To run the application, you use the interpreter and the class name:

```
$ java HelloWorld
Hello World
```

- The advantage of an IDE like NetBeans is that management of the class path, compilation, and running the Java application are handled through the tool.

Code Blocks

- Every class declaration is enclosed in a code block.
- Method declarations are enclosed in code blocks.
- Java fields and methods have block (or class) scope.
- Code blocks are defined in braces:

```
{  }
```

Example:

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

Primitive Data Types

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 7L 0xff 0b or 0B	3.0 22.0F .3337F 4.022E23	'a' '\u0061' '\n'	true false

Append uppercase or lowercase "`L`" or "`F`" to the number to specify a long or a float number.

Numeric Literals

- Any number of underscore characters (_) can appear between digits in a numeric field.
- This can improve the readability of your code.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Operators

- Simple assignment operator
  - = Simple assignment operator
- Arithmetic operators
  - + Additive operator (also used for String concatenation)
  - – Subtraction operator
  - * Multiplication operator
  - / Division operator
  - % Remainder operator
- Unary operators
  - + Unary plus operator; indicates positive
  - – Unary minus operator; negates an expression
  - ++ Increment operator; increments a value by 1
  - -- Decrement operator; decrements a value by 1
  - ! Logical complement operator; inverts the value of a boolean

Logical Operators

- Equality and relational operators

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

- Conditional operators

&&	Conditional-AND
||	Conditional-OR
?:	Ternary (shorthand for `if-then-else` statement)

- Type comparison operator

`instanceof`	Compares an object to a specified type

`if else` Statement

```
public class IfElse {

    public static void main(String args[]){
        long a = 1;
        long b = 2;

        if (a == b){
            System.out.println("True");
        } else {
           System.out.println("False");
       }

    }
}
```

`switch` Statement

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

       String color = "Blue";
       String shirt = " Shirt";

       switch (color){
           case "Blue":
               shirt = "Blue" + shirt;
               break;
           case "Red":
               shirt = "Red" + shirt;
               break;
           default:
               shirt = "White" + shirt;
       }

       System.out.println("Shirt type: " + shirt);
   }
}
```

`while` Loop

```
package com.example.review;

public class WhileTest {

    public static void main(String args[]) {
        int x = 10;
        while (x < 20) {
            System.out.print("value of x : " + x);
            x++;
            System.out.print("\n");
        }
    }
}
```

expression returning boolean value

do-while Loop

```
package com.example;

public class DoWhileTest {

    public static void main(String args[]) {

        int x = 30;

        do {
            System.out.print("value of x : " + x);
            x++;
            System.out.print("\n");
        } while (x < 20);
    }
}
```

expression returning boolean value

`for` Loop

```
public class ForLoop {

    public static void main(String args[]){

        for (int i = 0; i < 9;  i++ ){
            System.out.println("i: " + i);
        }

    }
}
```

Arrays and `for-each` Loop

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

        String[] names = new String[3];

        names[0] = "Blue Shirt";
        names[1] = "Red Shirt";
        names[2] = "Black Shirt";

        int[] numbers = {100, 200, 300};

        for (String name:names){
            System.out.println("Name: " + name);
        }

        for (int number:numbers){
            System.out.println("Number: " + number);
        }
    }
}
```

Arrays are objects. Array objects have a final field length.

Strings

```
public class Strings {

    public static void main(String args[]){

        char letter = 'a';

        String string1 = "Hello";
        String string2 = "World";
        String string3 = "";
       String dontDoThis = new String ("Bad Practice");

       string3 = string1 + string2; // Concatenate strings

       System.out.println("Output: " + string3 + " " + letter);

    }
}
```

String Operations: `StringBuilder`

```
public class StringOperations {

    public static void main(String arg[]) {

        StringBuilder sb = new StringBuilder("hello");
        System.out.println("string sb: " + sb);
        sb.append(" world");
        System.out.println("string sb: " + sb);

        sb.append("!").append(" are").append(" you?");
        System.out.println("string sb: " + sb);

        sb.insert(12, " How");
        System.out.println("string sb: " + sb);

        // Get length
        System.out.println("Length: " + sb.length());

        // Get SubString
        System.out.println("Sub: " + sb.substring(0, 5));
    }
}
```

A Simple Java Class: `Employee`

A Java class is often used to represent a concept.

```
package com.example.domain;
public class Employee {
    public int empId;
    public String name;
    public String ssn;
    public double salary;

    public Employee () {
    }

    public int getEmpId () {
      return empId;
    }
}
```

class declaration

fields

a constructor

a method

Methods

When a class has data fields, a common Lab is to provide methods for storing data (setter methods) and retrieving data (getter methods) from the fields.

```
package com.example.domain;
public class Employee {
    public int empId;
    // other fields...
    public void setEmpId(int empId) {
        this.empId = empId;
    }
    public int getEmpId() {
        return empId;
   }
   // getter/setter methods for other fields...
}
```

Often a pair of methods to set and get the current field value.

Creating an Instance of a Class

To construct or create an instance (object) of the `Employee` class, use the `new` keyword.

```
     /* In some other class, or a main method */
Employee emp = new Employee();
emp.empId = 101;   // legal if the field is public,
                   // but not good OO practice
emp.setEmpId(101); // use a method instead
emp.setName("John Smith");
emp.setSsn("011-22-3467");
emp.setSalary(120345.27);
```

Invoking an instance method.

- In this fragment of Java code, you construct an instance of the `Employee` class and assign the reference to the new object to a variable called `emp`.
- Then you assign values to the `Employee` object.

Constructors

```
public class Employee {
    public Employee() {
    }
      }
```

A simple no-argument (no-arg) constructor

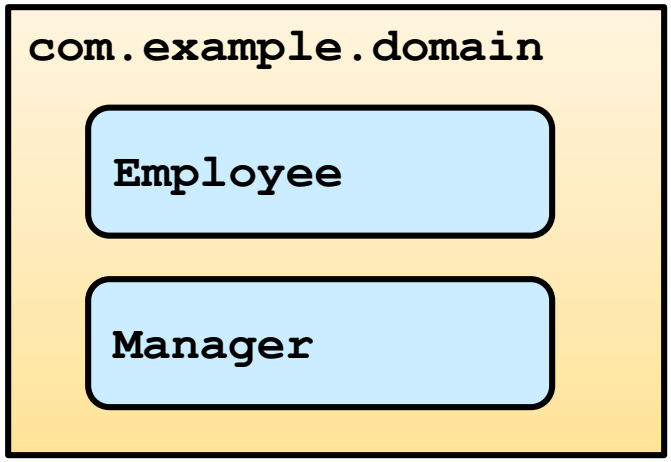
```
Employee emp = new Employee();
```

- A constructor is used to create an instance of a class.
- Constructors can take parameters.
- A constructor is declared with the same name as its class.

`package` Statement

- The `package` keyword is used in Java to group classes together.
- A package is implemented as a folder and, like a folder, provides a *namespace* to a class.

namespace view



folder view

```
+com
|_+example
  |_+domain
    |_+Employee.java
    |_+Manager.java
```

Always declare a package!

`import` Statements

The `import` keyword is used to identify classes you want to reference in your class.

- The `import` statement provides a convenient way to identify classes that you want to reference in your class.

```
import java.util.Date;
```

- You can import a single class or an entire package:

```
import java.util.*;
```

- You can include multiple `import` statements:

```
import java.util.Date;
import java.util.Calendar;
```

- It is good Lab to use the full package and class name rather than the wildcard `*` to avoid class name conflicts.

`import` Statements

- `import` statements follow the package declaration and precede the class declaration.
- An `import` statement is not required.
- By default, your class always imports `java.lang.*`
- You do not need to import classes that are in the same package:

```
package com.example.domain;
import com.example.domain.Manager; // unused import
```

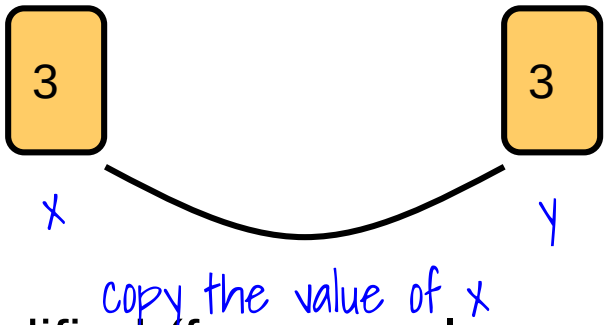
Java Is Pass-By-Value

The Java language (unlike C++) uses pass-by-value for all assignment operations.

- To visualize this with primitives, consider the following:

```
int x = 3;
int y = x;
```

- The value of `x` is copied and passed to `y`:



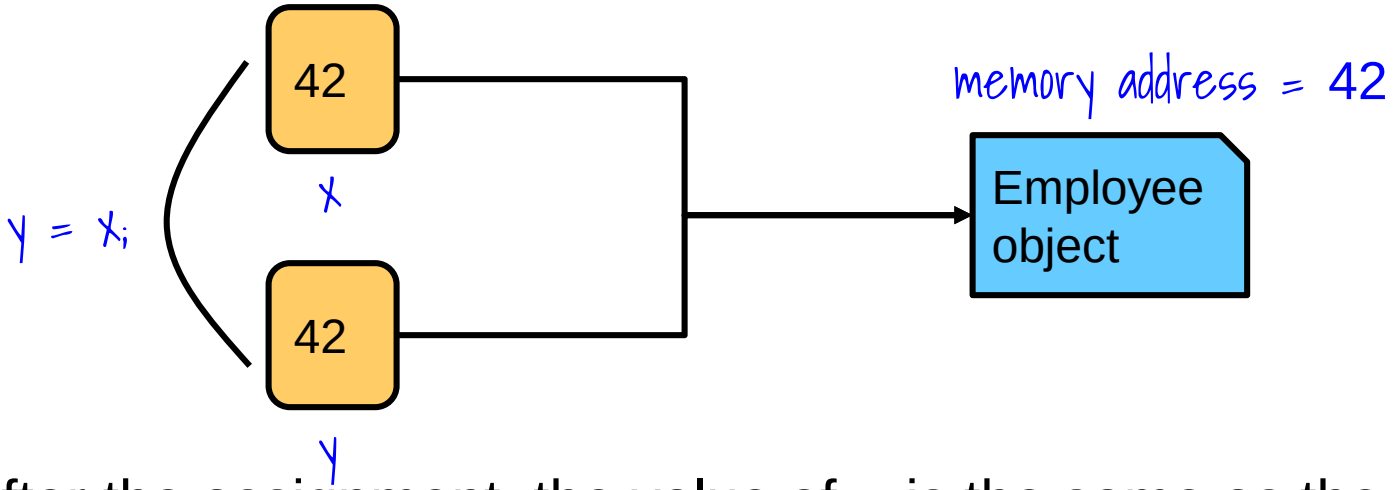
- If `x` is later modified (for example, `x = 5;`), the value of `y` remains unchanged.

Pass-By-Value for Object References

For Java objects, the *value* of the right side of an assignment is a reference to memory that stores a Java object.

```
Employee x = new Employee();
Employee y = x;
```

- The reference is some address in memory.



- After the assignment, the value of `y` is the same as the value of `x`: a reference to the same `Employee` object.

Objects Passed as Parameters

```
 4 public class ObjectPassTest {
 5     public static void main(String[] args) {
 6         ObjectPassTest test = new ObjectPassTest();
 7         Employee x = new Employee ();
 8         x.setSalary(120_000.00);
 9         test.foo(x);
10         System.out.println ("Employee salary: "
11           + x.getSalary());
12     }
13
14     public void foo(Employee e){
15       e.setSalary(130_000.00);
16       e = new Employee();
17       e.setSalary(140_000.00);
18     }
```

salary set to 120_000

What will `x.getSalary()` return at the end of the `main()` method?

Garbage Collection

When an object is instantiated by using the `new` keyword, memory is allocated for the object. The scope of an object reference depends on where the object is instantiated:

```
public void someMethod() {
    Employee e = new Employee();
    // operations on e
}
```

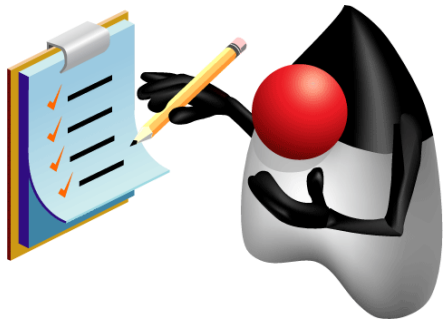
Object `e` scope ends here.

- When `someMethod` completes, the memory referenced by `e` is no longer accessible.
- Java's garbage collector recognizes when an instance is no longer accessible and eligible for collection.

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
  - Create primitive variables
  - Use Operators
  - Manipulate Strings
  - Use `if-else` and `switch` branching statements
  - Iterate with loops
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements



Lab 2-1 Overview: Creating Java Classes

This Lab covers the following topics:

- Creating a Java class using the NetBeans IDE
- Creating a Java class with a `main` method
- Writing code in the body of the `main` method to create an instance of the `Employee` object and print values from the class to the console
- Compiling and testing the application by using the NetBeans IDE

Quiz

Which is the printed result in the following fragment?

```
public float average (int[] values) {
    float result = 0;
    for (int i = 1; i < values.length; i++)
        result += values[i];
    return (result/values.length);
}
// ... in another method in the same class
int[] nums = {100, 200, 300};
System.out.println (average(nums));
```

a. 100.00
b. 150.00
c. 166.66667
d. 200.00

Quiz

In the following fragment, which two statements are false?

```
package com.oracle.test;
public class BrokenClass {
    public boolean valid = "false";
    public String s = "A new string";
    public int i = 40_000.00;
    public BrokenClass() { }
}
```

a. An `import` statement is missing.
b. The `boolean valid` is assigned a String.
c. `String s` is created.
d. `BrokenClass` method is missing a `return` statement.
e. You need to create a new `BrokenClass` object.
f. The integer value `i` is assigned a double.

Quiz

What is displayed when the following code snippet is compiled and executed?

```
String s1 = new String("Test");
String s2 = new String("Test");
if (s1==s2)
    System.out.println("Same");
if (s1.equals(s2))
    System.out.println("Equals");
```

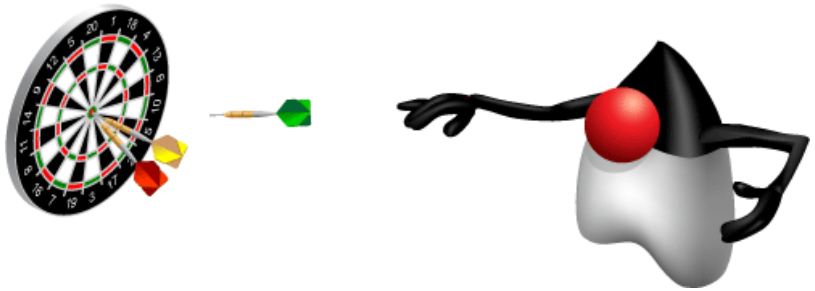
a. Same
b. Equals
c. Same Equals
d. Compiler Error

Encapsulation and Subclassing

Objectives

After completing this lesson, you should be able to do the following:

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods

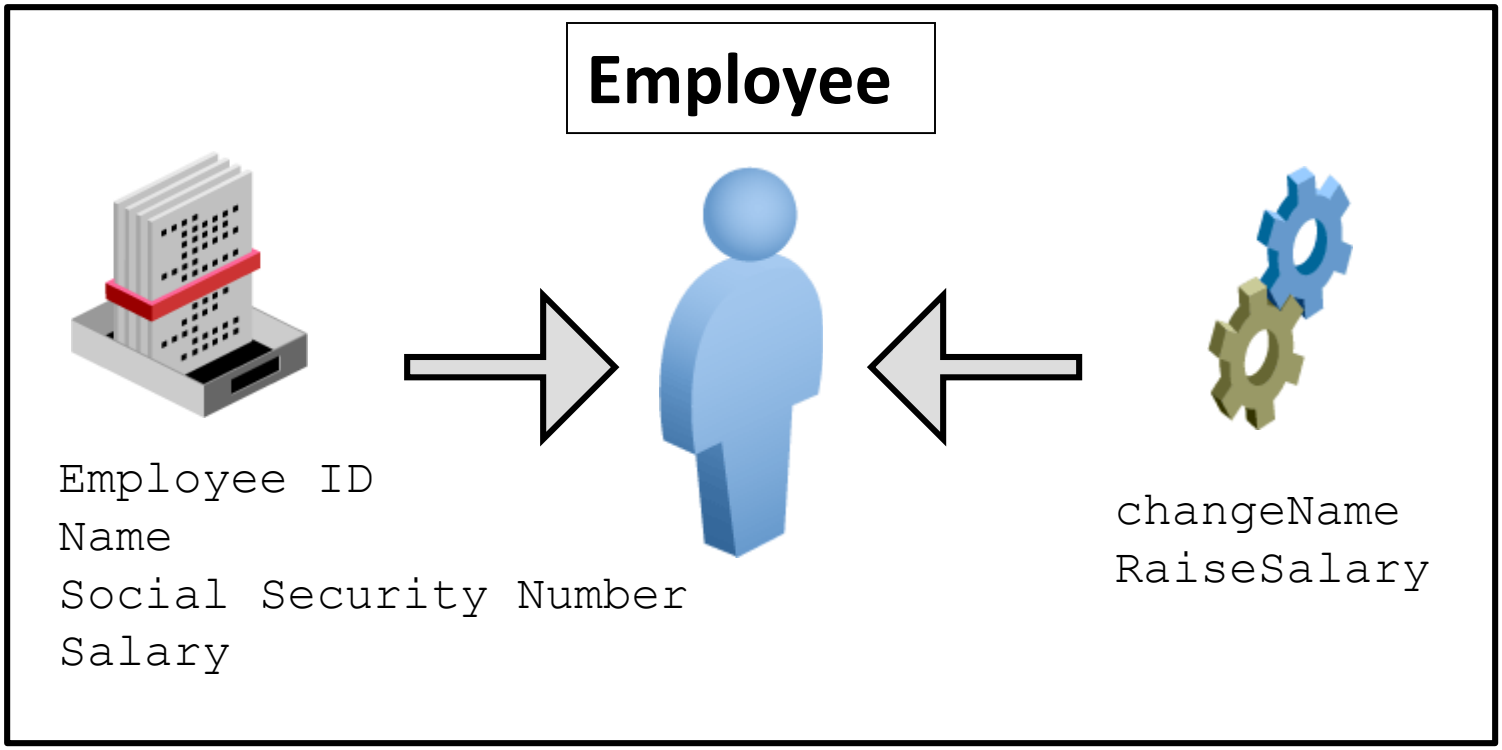


Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts. The other three are inheritance, polymorphism, and abstraction.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
  - Data variables, or fields, are hidden from the user of the object.
  - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
  - As long as the services do not change, the implementation can be modified without impacting the user.

Encapsulation: Example

What data and operations would you encapsulate in an object that represents an employee?



Encapsulation: Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
Employee emp=new Employee();
emp.salary=2000; //  Compiler error- salary is a private field
emp.raiseSalary(2000); //ok
```

- The `private` keyword can also be applied to a method to hide an implementation detail.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields `private`.

- The `Employee` class currently uses `public` access for all of its fields.
- To encapsulate the data, make the fields `private`.

```
public class Employee {

    private int empId;
    private String name;
    private String ssn;
    private double salary;

 //... constructor and methods
}
```

Declaring fields `private` prevents direct access to this data from a class instance.

```
// illegal!
emp.salary =
1_000_000_000.00;
```

`Employee` Class Refined

```
public class Employee {
    // private fields ...
    public Employee () {
    }
    // Remove all of the other setters
    public void changeName(String newName) {
         if (newName != null) {
             this.name = newName;
         }
    }

    public void raiseSalary(double increase) {
        this.salary += increase;
    }
}
```

Encapsulation step 2: These method names make sense in the context of an Employee.

Make Classes as Immutable as Possible

```
public class Employee {
   // private fields ...
   // Create an employee object
   public Employee (int empId, String name,
                    String ssn, double salary) {
       this.empId = empId;
       this.name = name;
       this.ssn = ssn;
      this.salary = salary;
   }

   public void changeName(String newName) { ... }

   public void raiseSalary(double increase) { ... }
}
```

Encapsulation step 3: Remove the no-arg constructor; implement a constructor to set the value of all fields.

Method Naming: Best Practices

Although the fields are now hidden by using `private` access, there are some issues with the current `Employee` class.

- The setter methods (currently `public` access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original `Employee` class design.
- Two best practices for methods:
  - Hide as many of the implementation details as possible.
  - Name the method in a way that clearly identifies its use or functionality.
- The original model for the `Employee` class had a Change Name and an Increase Salary operation.

Encapsulation: Benefits

The benefits of using encapsulation are as follows:

- Protects an object from unwanted access by clients
- Prevents assigning undesired values for its variables by the clients, which can make the state of an object unstable
- Allows changing the class implementation without modifying the client interface

Creating Subclasses

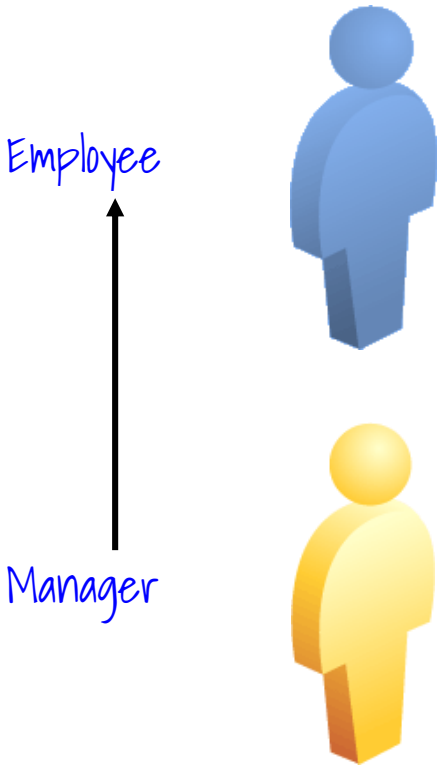
You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

```
package com.example.domain;
public class Manager {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    private String deptName;
    public Manager () {  }
    // access and mutator methods...
}
```

wait a minute...
this code looks very familiar....

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



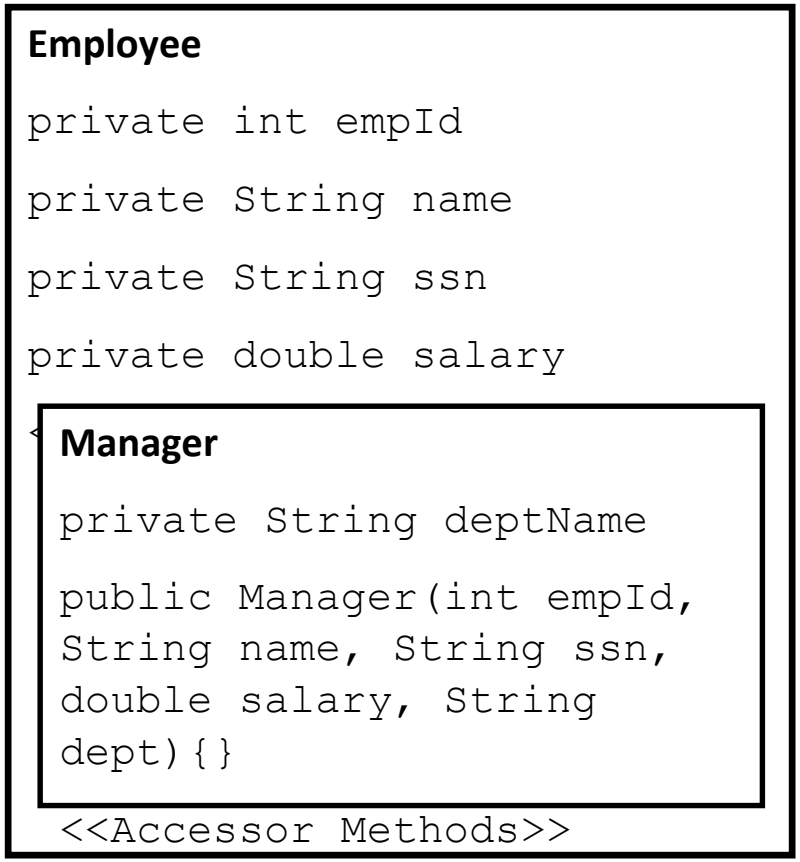
superclass: `Employee`
("parent" class)

subclass: `Manager`,
is an `Employee`
("child" class)

`Manager` Subclass

```
public class Manager extends Employee { }
```

The keyword **`extends`** creates the inheritance relationship:



Constructors in Subclasses

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
  - If you do not declare a constructor, a default no-arg constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.

Using `super`

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, `Manager` calls the constructor of `Employee`.
- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (nonprivate) field.

```
super (empId, name, ssn, salary);
```

Constructing a `Manager` Object

Creating a `Manager` object is the same as creating an `Employee` object:

```
Manager mgr = new Manager (102, "Barbara Jones",
                  "107-99-9078", 109345.67, "Marketing");
```

- All of the `Employee` methods are available to `Manager`:

```
mgr.raiseSalary (10000.00);
```

- The `Manager` class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```

Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
  - Argument lists *must* differ.
  - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)
public String print (int i)
```

Overloaded Constructors

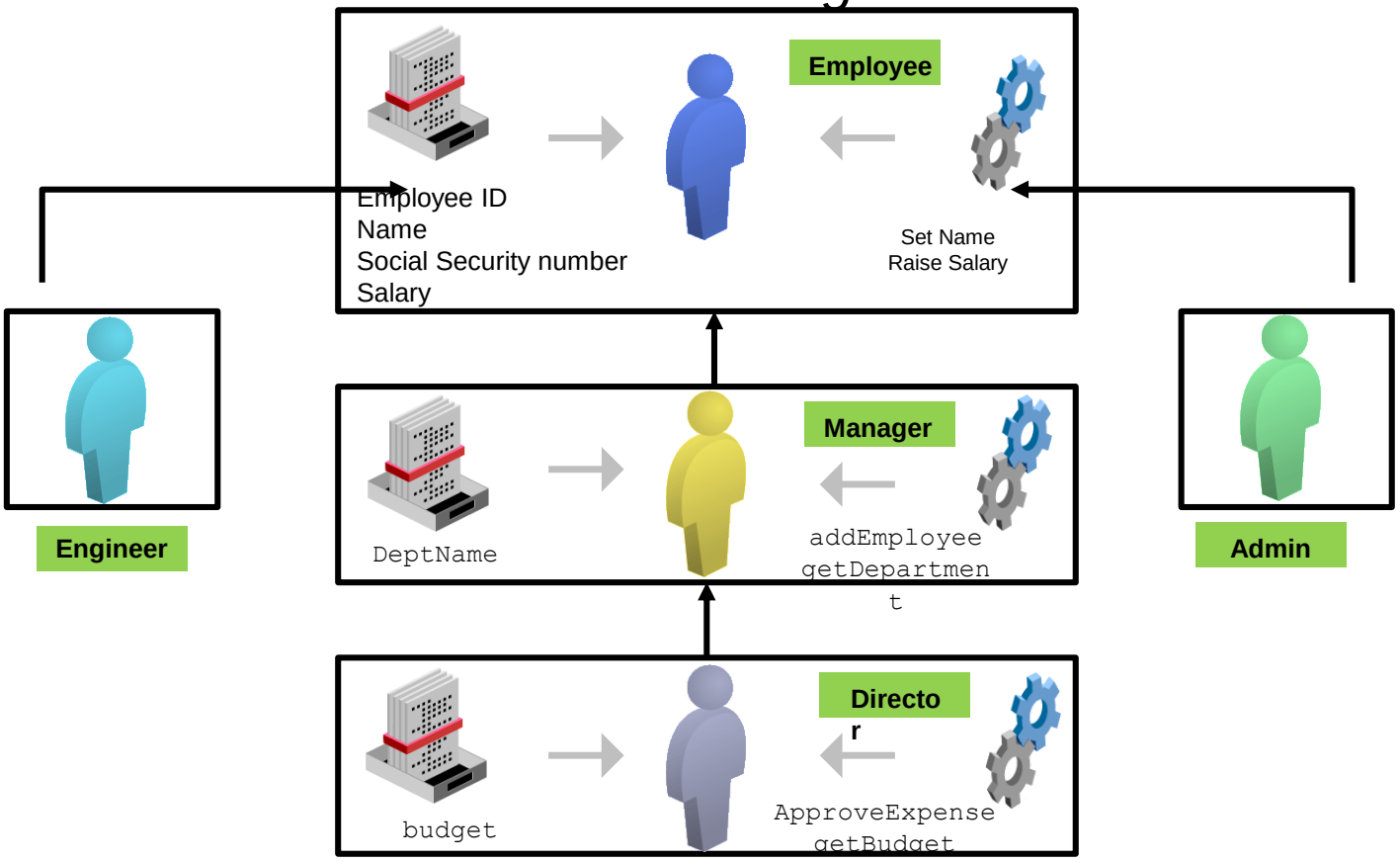
- In addition to overloading methods, you can overload constructors.
- The overloaded constructor is called based upon the parameters specified when the `new` is executed.

Overloaded Constructors: Example

```
public class Box {

    private double length, width, height;

    public Box() {
        this.length = 1;
        this.height = 1;
        this.width = 1;
    }

    public Box(double length) {
        this.width = this.length = this.height = length;
    }

    public Box(double length, double width, double height) {
        this.length = length;
        this.height = height;
        this.width = width;
        System.out.println("and the height of " + height + ".");
    }

    double volume() {
        return width * height * length;
    }
}
```

Single Inheritance

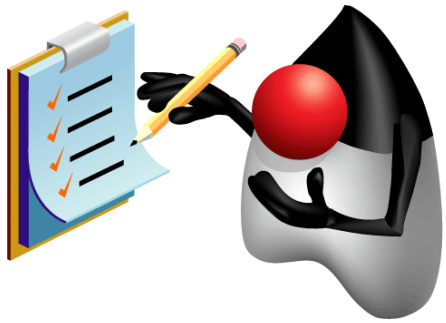
The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Summary

In this lesson, you should have learned how to:

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods



Lab 3-1 Overview:
Creating Subclasses

This Lab covers the following topics:

a. Applying encapsulation principles to the `Employee` class that you created in the previous practice
b. Creating subclasses of `Employee`, including `Manager`, `Engineer`, and Administrative assistant (`Admin`)
c. Creating a subclass of `Manager` called `Director`
d. Creating a test class with a `main` method to test your new classes

Quiz

Which of the following declarations demonstrates the application of good Java naming conventions?

a. `public class repeat { }`

b. `public void Screencoord (int x, int y){}`

c. `private int XCOORD;`

d. `public int calcOffset (int xCoord, int yCoord) { }`

Quiz

What changes would you perform to make this class immutable? (Choose all that apply.)

```
public class Stock {
    public String symbol;
                public double price;
    public int shares;
                public double getStockValue() { }
                public void setSymbol(String symbol) { }
    public void setPrice(double price) { }
    public void setShares(int number) { }
}
```

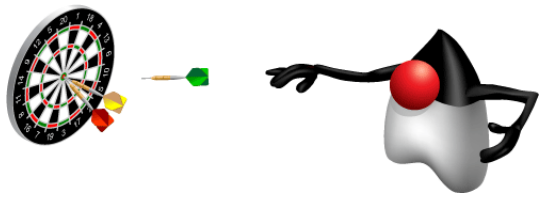
a. Make the fields `symbol`, `shares`, and `price` `private`.
b. Remove `setSymbol`, `setPrice`, and `setShares`.
c. Make the `getStockValue` method `private`.
d. Add a constructor that takes `symbol`, `shares`, and `price` as arguments.

Overriding Methods, Polymorphism, and Static Classes

Objectives

After completing this lesson, you should be able to do the following:

- Use access levels: `private`, `protected`, default, and `public`
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern



Using Access Control

- You have seen the keywords `public` and `private`.
- There are four access levels that can be applied to data fields and methods.
- Classes can be default (no modifier) or `public`.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Protected Access Control: Example

```
package demo;
public class Foo {
    protected int result = 20;
    int num= 25;
}
```

subclass-friendly declaration

```
package test;
import demo.Foo;
public class Bar extends Foo {
     private int sum = 10;
     public void reportSum () {
         sum += result;
         sum +=num;
     }
}
```

compiler error

Access Control: Good Practice

A good Lab when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
package demo;
public class Foo3 {
   private int result = 20;
   protected int getResult() {
    return this.result;
   }
}
```

```
package test;
import demo.Foo3;
public class Bar3 extends Foo3 {
    private int sum = 10;
    public void reportSum() {
        sum += getResult();
    }
}
```

Overriding Methods

Consider a requirement to provide a String that represents some details about the `Employee` class fields.

```
 3 public class Employee {
 4     private int empId;
 5     private String name;
14     // Lines omitted
15
16     public String getDetails() {
17       return "ID: " + empId + " Name: " + name;
18     }
```

Overriding Methods

In the `Manager` class, by creating a method with the same signature as the method in the `Employee` class, you are *overriding* the `getDetails` method:

```
 3 public class Manager extends Employee {
 4     private String deptName;
17     // Lines omitted
18
19     @Override
20     public String getDetails() {
21       return super.getDetails () +
22         " Dept: " + deptName;
23     }
```

A subclass can invoke a parent method by using the `super` keyword.

Invoking an Overridden Method

- Using the previous examples of `Employee` and `Manager`:

```
 5   public static void main(String[] args) {
 6     Employee e = new Employee(101, "Jim Smith",
 7         "011-12-2345", 100_000.00);
 8     Manager m = new Manager(102, "Joan Kern",
 9         "012-23-4567", 110_450.54, "Marketing");
10
11     System.out.println(e.getDetails());
12     System.out.println(m.getDetails());
13   }
```

- The correct `getDetails` method of each class is called:

```
ID: 101 Name: Jim Smith
ID: 102 Name: Joan Kern Dept: Marketing
```

Virtual Method Invocation

- What happens if you have the following?

```
 5   public static void main(String[] args) {
 6     Employee e = new Manager(102, "Joan Kern",
 7         "012-23-4567", 110_450.54, "Marketing");
 8
 9     System.out.println(e.getDetails());
10   }
```

- During execution, the object's runtime type is determined to be a `Manager` object:

```
ID: 102 Name: Joan Kern Dept: Marketing
```

- At run time, the method that is executed is referenced from a `Manager` object.
- This is an aspect of polymorphism called *virtual method invocation*.

Accessibility of Overriding Methods

The overriding method cannot be less accessible than the method in the parent class.

```
public class Employee {
    //... other fields and methods
    public String getDetails() { ... }
}
```

```
 3 public class BadManager extends Employee {
 4     private String deptName;
 5     // lines omitted
20     @Override
21     private String getDetails() { // Compile error
22       return super.getDetails () +
23         " Dept: " + deptName;
24     }
```

Applying Polymorphism

Suppose that you are asked to create a new class that calculates a bonus for employees based on their salary and their role (employee, manager, or engineer):

```
 3 public class BadBonus {
 4   public double getBonusPercent(Employee e){
 5     return 0.01;
 6   }
 7
 8   public double getBonusPercent(Manager m){
 9     return 0.03;
10   }
11
12   public double getBonusPercent(Engineer e){
13     return 0.01;
14   }
// Lines omitted
```

not very object-oriented!

Applying Polymorphism

A good Lab is to pass parameters and write methods that use the most generic possible form of your object.

```
public class GoodBonus {
  public static double getBonusPercent(Employee e){
     // Code here
  }
```

```
// In the Employee class
  public double calcBonus(){
    return this.getSalary() * GoodBonus.getBonusPercent(this);
  }
```

- One method will calculate the bonus for every type.

Using the `instanceof` Keyword

The Java language provides the `instanceof` keyword to determine an object's class type at run time.

```
 3 public class GoodBonus {
 4   public static double getBonusPercent(Employee e){
 5     if (e instanceof Manager){
 6       return 0.03;
 7     }else if (e instanceof Director){
 8       return 0.05;
 9     }else {
10       return 0.01;
11     }
12   }
13 }
```

Overriding Object methods

The root class of every Java class is `java.lang.Object`.

- All classes will subclass `Object` by default.
- You do not have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```

is equivalent to

```
public class Employee extends Object { //... }
```

- The root class contains several nonfinal methods, but there are three that are important to consider overriding:
  - `toString`, `equals`, and `hashCode`

`Object toString` Method

The `toString` method returns a String representation of the object.

```
Employee e = new Employee (101, "Jim Kern", ...)
System.out.println (e);
```

- You can use `toString` to provide instance information:

```
public String toString () {
    return "Employee id:  " + empId + "\n"+
           "Employee name:" + name;
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.

`Object equals` Method

The `Object equals` method compares only object references.

- If there are two objects x and y in any class, x is equal to y if and only if x and y refer to the same object.
- Example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Because what we really want is to test the contents of the Employee object, we need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```

Overriding `equals` in `Employee`

An example of overriding the `equals` method in the `Employee` class compares every field for equality:

```
@Override
public boolean equals (Object o) {
    boolean result = false;
    if ((o != null) && (o instanceof Employee)) {
        Employee e = (Employee)o;
        if ((e.empId == this.empId) &&
            (e.name.equals(this.name)) &&
            (e.ssn.equals(this.ssn)) &&
            (e.salary == this.salary)) {
               result = true;
        }
    }     return result;
}
```

Overriding `Object` `hashCode`

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer hashcode returned for the two objects should also be equal.

```
@Override //generated by NetBeans
public int hashCode() {
    int hash = 7;
    hash = 83 * hash + this.empId;
    hash = 83 * hash + Objects.hashCode(this.name);
    hash = 83 * hash + Objects.hashCode(this.ssn);
    hash = 83 * hash + (int)
(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
    return hash;
}
```

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

The varargs notation treats the `nums` parameter as an array.

```
public class Statistics {
    public float average(int... nums) {
        int sum = 0;
        for (int x : nums) {  // iterate int array nums
            sum += x;
        }
        return ((float) sum / nums.length);
    }
}
```

- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.

Casting Object References

After using the `instanceof` operator to verify that the object you received as an argument is a subclass, you can access the full functionality of the object by casting the reference:

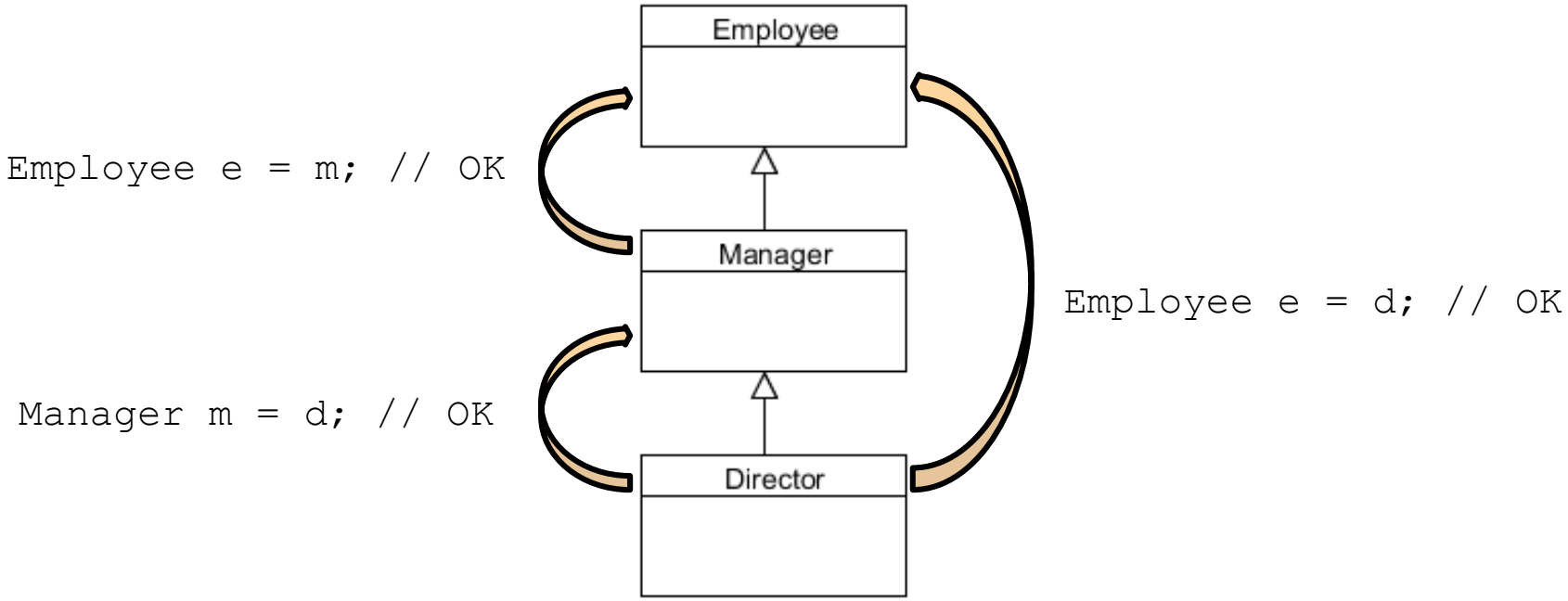
```
 4   public static void main(String[] args) {
 5     Employee e = new Manager(102, "Joan Kern",
 6         "012-23-4567", 110_450.54, "Marketing");
 7
 8     if (e instanceof Manager){
 9       Manager m = (Manager) e;
10       m.setDeptName("HR");
11       System.out.println(m.getDetails());
12     }
13   }
```

Without the cast to `Manager`, the `setDeptName` method would not compile.

Upward Casting Rules

Upward casts are always permitted and do not require a cast operator.

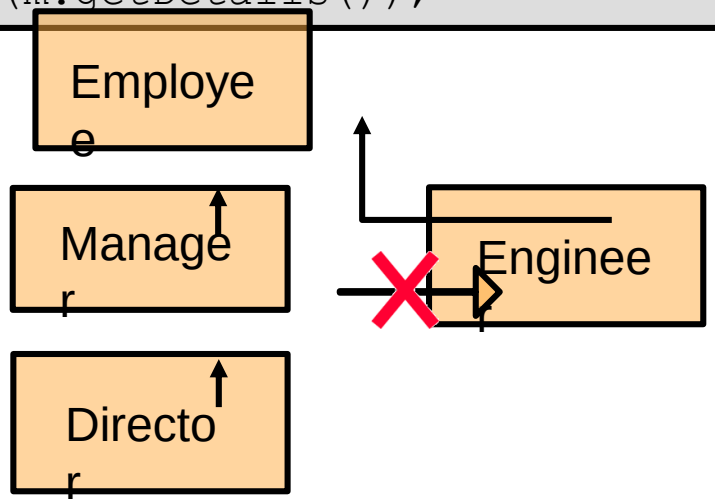
```
Director d = new Director();
Manager m = new Manager();
```



Downward Casting Rules

For downward casts, the compiler must be satisfied that the cast is possible.

```
 5      Employee e = new Manager(102, "Joan Kern",
 6          "012-23-4567", 110_450.54, "Marketing");
 7
 8      Manager m = (Manager)e; // ok
 9      Engineer eng = (Manager)e; // Compile error
10      System.out.println(m.getDetails());
```



`static` Keyword

The `static` modifier is used to declare fields and methods as class-level resources.

Static class members:

- Can be used without object instances
- Are used when a problem is best solved without objects
- Are used when objects of the same type need to share fields
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason

Static Methods

Static methods are methods that can be called even if the class they are declared in has not been instantiated.

Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented
  - `java.lang.Math` contains many static methods
- Are commonly used in place of constructors to perform tasks related to object initialization
- Cannot access nonstatic members within the same class

Using Static Variables and Methods: Example

```
 3 public class A01MathTest {
 4   public static void main(String[] args) {
 5     System.out.println("Random: " + Math.random() * 10);
 6     System.out.println("Square root: " + Math.sqrt(9.0));
 7     System.out.println("Rounded random: " +
 8         Math.round(Math.random()*100));
 9     System.out.println("Abs: " + Math.abs(-9));
10   }
11 }
```

Implementing Static Methods

- Use the static keyword before the method
- The method has parameters and return types like normal

```
 3 import java.time.LocalDate;
 4
 5 public class StaticHelper {
 6
 7     public static void printMessage(String message) {
 8         System.out.println("Messsage for " +
 9          LocalDate.now() + ": " + message);
10     }
11
12 }
```

Calling Static Methods

```
double d = Math.random();
StaticHelper.printMessage("Hello");
```

When calling static methods, you should:

- Qualify the location of the method with a class name if the method is located in a different class than the caller
  - Not required for methods within the same class
- Avoid using an object reference to call a static method

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated.

Static variables are:

- Called class variables
- Limited to a single copy per JVM
- Useful for containing shared data
  - Static methods store data in static variables.
  - All object instances share a single copy of any static variables.
- Initialized when the containing class is first loaded

Defining Static Variables

```
 4 public class StaticCounter {
 5     private static int counter = 0;
 6
 7     public static int getCount() {
 8         return counter;
 9     }
10
11     public static void increment(){
12       counter++;
13     }
14 }
```

Only one copy in memory

Using Static Variables

```
double p = Math.PI;
```

```
 5   public static void main(String[] args) {
 6     System.out.println("Start: " + StaticCounter.getCount());
 7     StaticCounter.increment();
 8     StaticCounter.increment();
 9     System.out.println("End: " + StaticCounter.getCount());
10}
```

When accessing static variables, you should:

- Qualify the location of the variable with a class name if the variable is located in a different class than the caller
  - Not required for variables within the same class
- Avoid using an object reference to access a static variable

Static Initializers

- Static initializer block is a code block prefixed by the `static` keyword.

```
 3 public class A04StaticInitializerTest {
 4   private static final boolean[] switches = new boolean[5];
 5
 6   static{
 7     System.out.println("Initializing...");
 8     for (int i=0; i<5; i++){
 9       switches[i] = true;
10     }
11   }
12
13   public static void main(String[] args) {
14     switches[1] = false; switches[2] = false;
15     System.out.print("Switch settings: ");
16     for (boolean curSwitch:switches){
17       if (curSwitch){System.out.print("1");}
18       else {System.out.print("0");}
19     }
```

static initialization block

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- Calling the `Math.random()` method can be written as:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```

Design Patterns

Design patterns are:

- Reusable solutions to common software development problems
- Documented in pattern catalogs
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the “Gang of Four”)
- A vocabulary used to discuss design

Singleton Pattern

The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {
    private static final SingletonClass instance =
            new SingletonClass();

    private SingletonClass() {}

    public static SingletonClass getInstance() {
        return instance;
    }
}
```

1

2

3

Singleton: Example

```
 3 public final class DbConfigSingleton {
 4   private final String hostName;
 5   private final String dbName;
 6   //Lines omitted
10   private static final DbConfigSingleton instance =
11             new DbConfigSingleton();
12
13   private DbConfigSingleton(){
14     // Values loaded from file in practice
15     hostName = "dbhost.example.com";
16     // Lines omitted
20   }
21
22   public static DbConfigSingleton getInstance() {
23     return instance;
24   }
```

Immutable Classes

Immutable class:

- It is a class whose object state cannot be modified once created.
- Any modification of the object will result in another new immutable object.
- Example: Objects of `Java.lang.String`, any change on existing string object will result in another string; for example, replacing a character or creating substrings will result in new objects.

Example: Creating Immutable class in Java

```
public final class Contacts {
 private final String firstName;
 private final String lastName;

 public Contacts(String fname,String lname) {
     this.firstName= fname;
     this.lastName = lname;

 }
 public String getFirstName() {
     return firstName;
 }
 public String getLastName() {
     return lastName;
 }

public String toString() {
 return firstName +" - "+ lastName +" - "+ lastName;

}
}
```

Summary

In this lesson, you should have learned how to:

- Use access levels: `private`, `protected`, default, and `public`.
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern

Lab 4-1 Overview: Overriding Methods and Applying Polymorphism

This Lab covers the following topics:

- Modifying the `Employee`, `Manager`, and `Director` classes; overriding the `toString()` method
- Creating an `EmployeeStockPlan` class with a grant stock method that uses the `instanceof` keyword

Lab 4-2 Overview:
Overriding Methods and Applying Polymorphism

This Lab covers the following topics:

- Fixing compilation errors caused due to casting
- Identifying runtime exception caused due to improper casting

Lab 4-3 Overview:
Applying the Singleton Design Pattern

This Lab covers using the `static` and `final` keywords and refactoring an existing application to implement the singleton design pattern.

Quiz

Suppose that you have an `Account` class with a `withdraw()` method, and a `Checking` class that extends `Account` that declares its own `withdraw()` method. What is the result of the following code fragment?

```
Account acct = new Checking();
acct.withdraw(100);
```

a. The compiler complains about line 1.
b. The compiler complains about line 2.
c. Runtime error: incompatible assignment (line 1)
d. Executes `withdraw` method from the `Account` class
- Executes `withdraw` method from the `Checking` class

Quiz

Suppose that you have an `Account` class and a `Checking` class that extends `Account`. The body of the `if` statement in line 2 will execute.

```
Account acct = new Checking();
if (acct instanceof Checking) { // will this block run? }
```

a. True

b. False

Quiz

Suppose that you have an `Account` class and a `Checking` class that extends `Account`. You also have a `Savings` class that extends `Account`. What is the result of the following code?

```
Account acct1 = new Checking();
Account acct2 = new Savings();
Savings acct3 = (Savings)acct1;
```

a. `acct3` contains the reference to `acct1`.

b. A runtime `ClassCastException` occurs.

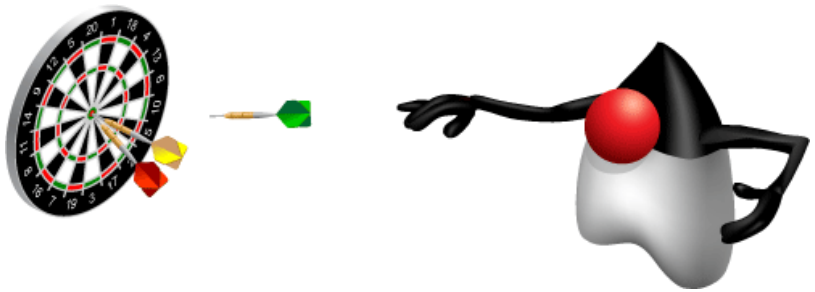
- The compiler complains about line 2.
- The compiler complains about the cast in line 3.

Abstract and Nested Classes

Objectives

After completing this lesson, you should be able to do the following:

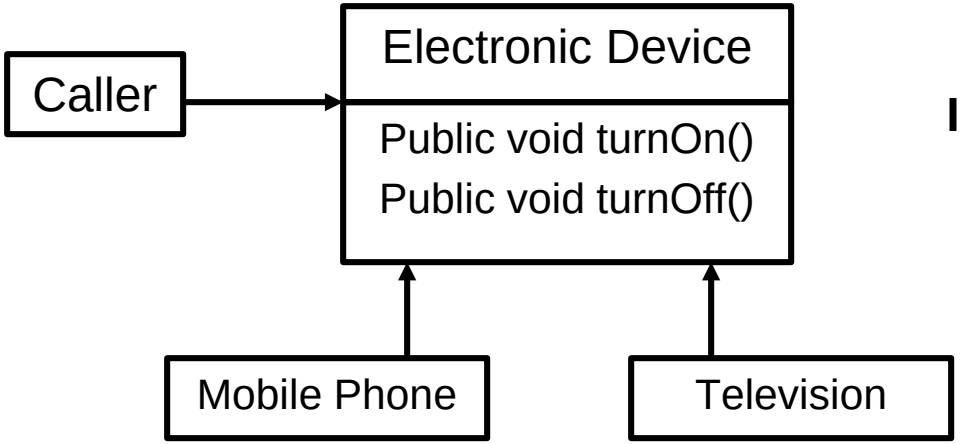
- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes



Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- **Method inheritance:** Subclasses avoid code duplication by inheriting method implementations.
- **Generalization:** Code that is designed to rely on the most generic type possible is easier to maintain.



Class Inheritance Diagram

Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized.

```
public class Television extends ElectronicDevice {

    public void turnOn() {
        changeChannel(1);
        initializeScreen();
    }
    public void turnOff() {}

    public void changeChannel(int channel) {}
    public void initializeScreen() {}

}
```

Defining Abstract Classes

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class **cannot** be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```

Defining Abstract Methods

A method can be declared as abstract by using the `abstract` method-level modifier.

```
public abstract class ElectronicDevice {

        public abstract void turnOn();
        public abstract void turnOff();
}
```

No braces

An abstract method:

- Cannot have a method body
- Must be declared in an abstract class
- Is overridden in subclasses

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and nonabstract methods.
- When inheriting from an abstract class, you must do either of the following:
  - Declare the child class as abstract.
  - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override
abstract method turnOn() in ElectronicDevice
```

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

Final Variables

The `final` modifier can be applied to variables.

Final variables may not change their values after they are initialized.

Final variables can be:

- Class fields
  - Final fields with compile-time constant expressions are constant variables.
  - Static can be combined with final to create an always-available, never-changing variable.
- Method parameters
- Local variables

Note: Final references must always reference the same object, but the contents of that object may be modified.

Declaring Final Variables

```
public class VariableExampleClass {
    private final int field;
    public static final int JAVA_CONSTANT = 10;

    public VariableExampleClass() {
        field = 100;
    }

    public void changeValues(final int param) {
        param = 1; // compile-time error
        final int localVar;
        localVar = 42;
        localVar = 43; // compile-time error
    }
}
```

Nested Classes

A nested class is a class declared within the body of another class. Nested classes:

- Have multiple categories
  - **Inner classes**
    - Member classes
    - Local classes
    - Anonymous classes
  - **Static nested classes**
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class

Example: Member Class

```
public class BankEMICalculator {
   private  String CustomerName;
   private  String AccountNo;
   private  double loanAmount;
   private  double monthlypayment;
   private EMICalculatorHelper helper = new EMICalculatorHelper();

    /*Setters ad Getters*/

      private class EMICalculatorHelper {
        int loanTerm = 60;
        double interestRate = 0.9;
        double interestpermonth=interestRate/loanTerm;

    protected double calcMonthlyPayment(double loanAmount)
      {
       double EMI= (loanAmount * interestpermonth) / ((1.0) - ((1.0) /
    Math.pow(1.0 + interestpermonth, loanTerm)));
      return(Math.round(EMI));
      }
   }
}
```

Inner class, `EMICalculatorHelper`

Enumerations

Java includes a typesafe enum to the language.

Enumerations (enums):

- Are created by using a variation of a Java class
- Provide a compile-time range check

```
public enum PowerState {
   OFF,
   ON,
   SUSPEND;
}
```

These are references to the only three `PowerState` objects that can exist.

An enum can be used in the following way:

```
Computer comp = new Computer();
comp.setState(PowerState.SUSPEND);
```

This method takes a `PowerState` reference .

Enum Usage

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {
    switch(state) {
        case OFF:
        //...
    }
}
```

PowerState.OFF

Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {
    OFF("The power is off"),
    ON("The usage power is high"),
    SUSPEND("The power usage is low");

    private String description;
    private PowerState(String d) {
        description = d;
    }
    public String getDescription() {
        return description;
    }
}
```

Call a `PowerState` constructor to initialize the `public static final OFF` reference.

The constructor may not be `public` or `protected`.

Complex Enums

- Here is the complex enum in action.

```
public class ComplexEnumsMain {

    public static void main(String[] args) {
        Computer comp = new Computer();
        comp.setState(PowerState.SUSPEND);
        System.out.println("Current state: " +
    comp.getState());
        System.out.println("Description: " +
    comp.getState().getDescription());
    }
}
```

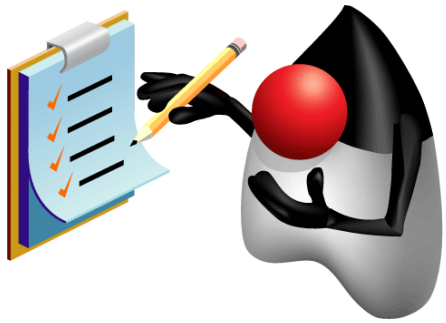
- Output

```
Current state: SUSPEND
Description: The power usage is low
```

Summary

In this lesson, you should have learned how to:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes



Lab 5-1 Overview:
Applying the Abstract Keyword

This Lab covers the following topics:

- Identifying potential problems that can be solved using abstract classes
- Refactoring an existing Java application to use abstract classes and methods

Lab 5-2 Overview:
Using Inner Class As a Helper Class

This Lab covers using an inner class as a helper class to perform some calculations in an Employee class.

Lab 5-3 Overview:
Using Java Enumerations

This Lab covers taking an existing application and refactoring the code to use an enum.

Quiz

Which two of the following should an abstract method not have to compile successfully?

a. A return value
b. A method implementation
c. Method parameters
d. `private` access

Quiz

Which of the following nested class types are inner classes?

a. Anonymous
b. Local
c. Static
d. Member

Quiz

A final field (instance variable) can be assigned a value either when declared or in all constructors.

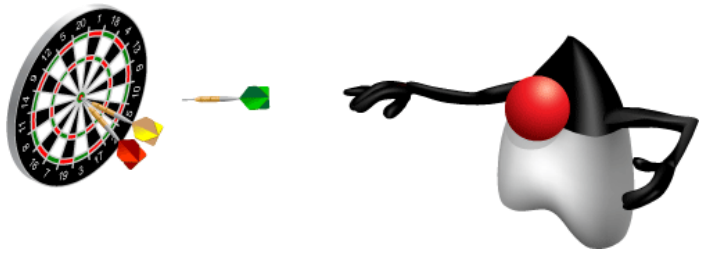
a. True
b. False

Interfaces and Lambda Expressions

Objectives

After completing this lesson, you should be able to do the following:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a lambda expression



Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
  - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns

A Problem Solved by Interfaces

Given: A company sells an assortment of products, very different from each other, and needs a way to access financial data in a similar manner.

- Products include:
  - Crushed Rock
    - Measured in pounds
  - Red Paint
    - Measured in gallons
  - Widgets
    - Measured by Quantity
- Need to calculate per item
  - Sales price
  - Cost
  - Profit

`CrushedRock` Class

- The `CrushedRock` class before interfaces

```
public class CrushedRock {
  private String name;
  private double salesPrice = 0;
  private double cost = 0;
  private double weight = 0; // In pounds

  public CrushedRock(double salesPrice, double cost,
double weight){
    this.salesPrice = salesPrice;
    this.cost = cost;
    this.weight = weight;
  }
}
```

The `SalesCalcs` Interface

- The `SalesCalcs` interface specifies the types of calculations required for our products.
  - Public, top-level interfaces are declared in their own `.java` file.

```
public interface SalesCalcs {
  public String getName();
  public double calcSalesPrice();
  public double calcCost();
  public double calcProfit();
}
```

Adding an Interface

- The updated `CrushedRock` class implements `SalesCalcs`.

```
public class CrushedRock implements SalesCalcs{
  private String name = "Crushed Rock";
...  // a number of lines not shown
  @Override
  public double calcCost(){
    return this.cost * this.weight;
  }

  @Override
  public double calcProfit(){
    return this.calcSalesPrice() - this.calcCost();
  }
}
```

Interface References

- Any class that implements an interface can be referenced by using that interface.
- Notice how the `calcSalesPrice` method can be referenced by the `CrushedRock` class or the `SalesCalcs` interface.

```
CrushedRock rock1 = new CrushedRock(12, 10, 50);
SalesCalcs rock2 = new CrushedRock(12, 10, 50);
System.out.println("Sales Price: " +
rock1.calcSalesPrice());
System.out.println("Sales Price: " +
rock2.calcSalesPrice());
```

- Output

```
Sales Price: 600.0
Sales Price: 600.0
```

Interface Reference Usefulness

- Any class implementing an interface can be referenced by using that interface. For example:

```
SalesCalcs[] itemList = new SalesCalcs[5];
ItemReport report = new ItemReport();

itemList[0] = new CrushedRock(12.0, 10.0, 50.0);
itemList[1] = new CrushedRock(8.0, 6.0, 10.0);
itemList[2] = new RedPaint(10.0, 8.0, 25.0);
itemList[3] = new Widget(6.0, 5.0, 10);
itemList[4] = new Widget(14.0, 12.0, 20);

System.out.println("==Sales Report==");
for(SalesCalcs item:itemList){
  report.printItemData(item);
}
```

Interface Code Flexibility

- A utility class that references the interface can process any implementing class.

```
public class ItemReport {
  public void printItemData(SalesCalcs item){
    System.out.println("--" + item.getName() + " Report--");
    System.out.println("Sales Price: " + item.calcSalesPrice());
    System.out.println("Cost: " + item.calcCost());
    System.out.println("Profit: " + item.calcProfit());
  }
}
```

`default` Methods in Interfaces

Java 8 has added `default` methods as a new feature:

```
public interface SalesCalcs {
... // A number of lines omitted
  public default void printItemReport(){
    System.out.println("--" + this.getName() + " Report--");
    System.out.println("Sales Price: " + this.calcSalesPrice());
    System.out.println("Cost: " + this.calcCost());
    System.out.println("Profit: " + this.calcProfit());
  }
}
```

`default` methods:

- Are declared by using the keyword `default`
- Are fully implemented methods within an interface
- Provide useful inheritance mechanics

`default` Method: Example

Here is an updated version of the item report using `default` methods.

```
SalesCalcs[] itemList = new SalesCalcs[5];

itemList[0] = new CrushedRock(12, 10, 50);
itemList[1] = new CrushedRock(8, 6, 10);
itemList[2] = new RedPaint(10, 8, 25);
itemList[3] = new Widget(6, 5, 10);
itemList[4] = new Widget(14, 12, 20);

System.out.println("==Sales Report==");
for(SalesCalcs item:itemList){
  item.printItemReport();
}
```

`static` Methods in Interfaces

Java 8 allows `static` methods in an interface. So it is possible to create helper methods like the following.

```
public interface SalesCalcs {
... // A number of lines omitted
  public static void printItemArray(SalesCalcs[] items){
    System.out.println(reportTitle);
      for(SalesCalcs item:items){
        System.out.println("--" + item.getName() + " Report--");
        System.out.println("Sales Price: " +
item.calcSalesPrice());
        System.out.println("Cost: " + item.calcCost());
        System.out.println("Profit: " + item.calcProfit());
      }
}
```

Constant Fields

Interfaces can have constant fields.

```
public interface SalesCalcs {
  public static final String reportTitle="\n==Static
   List Report==";
... // A number of lines omitted
```

Extending Interfaces

- Interfaces can extend interfaces:

```
public interface WidgetSalesCalcs extends SalesCalcs{
  public String getWidgetType();
}
```

- So now any class implementing `WidgetSalesCalc` must implement all the methods of `SalesCalcs` in addition to the new method specified here.

Implementing and Extending

- Classes can extend a parent class and implement an interface:

```
public class WidgetPro extends Widget implements
WidgetSalesCalcs{
  private String type;

  public WidgetPro(double salesPrice, double cost, long
quantity, String type){
    super(salesPrice, cost, quantity);
    this.type = type;
  }

  public String getWidgetType(){
    return type;
  }
}
```

Anonymous Inner Classes

- Define a class in place instead of in a separate file
- Why would you do this?
  - Logically group code in one place
  - Increase encapsulation
  - Make code more readable
- `StringAnalyzer` interface

```
public interface StringAnalyzer {
  public boolean analyze(String target, String
    searchStr);
}
```

- A single method interface
  - **Functional Interface**
- Takes two strings and returns a `boolean`

Anonymous Inner Class: Example

- Example method call with concrete class

```
20      // Call concrete class that implments StringAnalyzer
21      ContainsAnalyzer contains = new ContainsAnalyzer();
22
23      System.out.println("===Contains===");
24      Z03Analyzer.searchArr(strList01, searchStr, contains);
```

- Anonymous inner class example

```
22      Z04Analyzer.searchArr(strList01, searchStr,
23        new StringAnalyzer(){
24          @Override
25          public boolean analyze(String target, String
    searchStr){
26            return target.contains(searchStr);
27          }
28  });
```

- The class is created in place.

String Analysis Regular Class

- Class analyzes an array of strings given a search string
  - Print strings that contain the search string
  - Other methods could be written to perform similar string test
- Regular Class Example method

```
1 package com.example;
2
3 public class AnalyzerTool {
4   public boolean arrContains(String sourceStr, String
    searchStr){
5     return sourceStr.contains(searchStr);
6   }
7 }
8
```

String Analysis Regular Test Class

- Here is the code to test the class, `Z01Analyzer`

```
 4   public static void main(String[] args) {
 5     String[] strList =
 6     {"tomorrow","toto","to","timbukto","the","hello","heat"};
 7     String searchStr = "to";
 8     System.out.println("Searching for: " + searchStr);
 9
10     // Create regular class
11     AnalyzerTool analyzeTool = new AnalyzerTool();
12
13     System.out.println("===Contains===");
14     for(String currentStr:strList){
15       if  (analyzeTool.arrContains(currentStr, searchStr)){
16         System.out.println("Match: " + currentStr);
17       }
18     }
19 }
```

String Analysis Interface: Example

- What about using an interface?

```
3 public interface StringAnalyzer {
4    public boolean analyze(String sourceStr, String
     searchStr);
5 }
```

- `StringAnalyzer` is a single method functional interface.
- Replacing the previous example and implementing the interface looks like this:

```
 3 public class ContainsAnalyzer implements StringAnalyzer {
 4    @Override
 5    public boolean analyze(String target, String searchStr){
 6       return target.contains(searchStr);
 7    }
 8 }
```

String Analyzer Interface Test Class

```
 4   public static void main(String[] args) {
 5     String[] strList =
 6     {"tomorrow","toto","to","timbukto","the","hello","heat"};
 7     String searchStr = "to";
 8     System.out.println("Searching for: " + searchStr);
 9
10     // Call concrete class that implments StringAnalyzer
11     ContainsAnalyzer contains = new ContainsAnalyzer();
12
13     System.out.println("===Contains===");
14     for(String currentStr:strList){
15       if (contains.analyze(currentStr, searchStr)){
16         System.out.println("Match: " + currentStr);
17       }
18     }
19   }
```

Encapsulate the for Loop

- An improvement to the code is to encapsulate the `forloop`:

```
 3 public class Z03Analyzer {
 4
 5   public static void searchArr(String[] strList, String
    searchStr, StringAnalyzer analyzer){
 6     for(String currentStr:strList){
 7       if (analyzer.analyze(currentStr, searchStr)){
 8         System.out.println("Match: " + currentStr);
 9       }
10     }
11   }
// A number of lines omitted
```

String Analysis Test Class with Helper Method

- With the helper method, the main method shrinks to this:

```
13   public static void main(String[] args) {
14     String[] strList01 =
15     {"tomorrow","toto","to","timbukto","the","hello","heat"};
16     String searchStr = "to";
17     System.out.println("Searching for: " + searchStr);
18
19     // Call concrete class that implments StringAnalyzer
20     ContainsAnalyzer contains = new ContainsAnalyzer();
21
22     System.out.println("===Contains===");
23     Z03Analyzer.searchArr(strList01, searchStr, contains);
24   }
```

String Analysis Anonymous Inner Class

- Create anonymous inner class for third argument.

```
19      // Implement anonymous inner class
20      System.out.println("===Contains===");
21      Z04Analyzer.searchArr(strList01, searchStr,
22        new StringAnalyzer(){
23          @Override
24          public boolean analyze(String target, String
    searchStr){
25            return target.contains(searchStr);
26          }
27        });
28    }
```

String Analysis Lambda Expression

- Use lambda expression for the third argument.

```
13   public static void main(String[] args) {
14     String[] strList =
15     {"tomorrow","toto","to","timbukto","the","hello","heat"};
16     String searchStr = "to";
17     System.out.println("Searching for: " + searchStr);
18
19     // Lambda Expression replaces anonymous inner class
20     System.out.println("==Contains==");
21     Z05Analyzer.searchArr(strList, searchStr,
22       (String target, String search) -> target.contains(search));
23   }
```

Lambda Expression Defined

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

Basic Lambda examples

```
(int x, int y) -> x + y
(x, y) -> x + y

(x, y) -> { system.out.println(x + y);}

(String s) -> s.contains("word")
s -> s.contains("word")
```

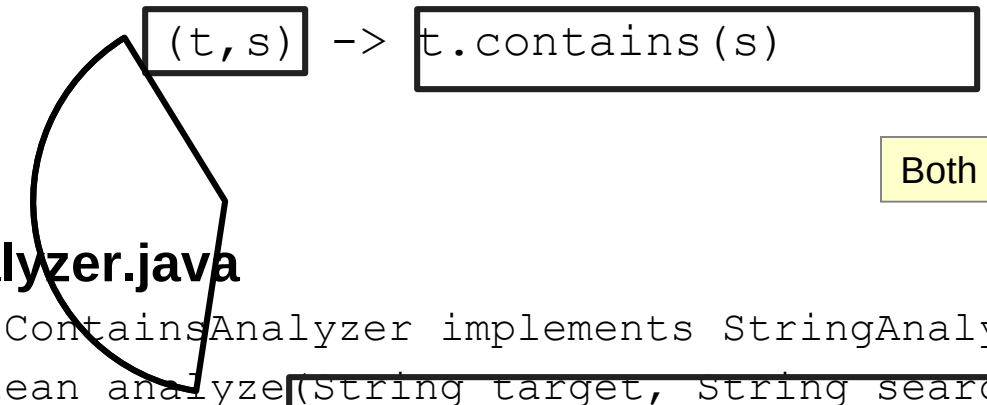
What Is a Lambda Expression?

```
(t,s) -> t.contains(s)
```

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {
  public boolean analyze(String target, String searchStr){
    return target.contains(searchStr);
  }
}
```

What Is a Lambda Expression?



ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {
  public boolean analyze(String target, String searchStr) {
    return target.contains(searchStr);
  }
}
```

What Is a Lambda Expression?

```
(t,s) -> t.contains(s)
```

Both have parameters

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {
  public boolean analyze(String target, String searchStr) {
    return target.contains(searchStr);
  }
}
```

Both have a body with one or more statements

Lambda Expression Shorthand

- Lambda expressions using shortened syntax

```
20      // Use short form Lambda
21      System.out.println("==Contains==");
22      Z06Analyzer.searchArr(strList01, searchStr,
23        (t, s) -> t.contains(s));
24
25      // Changing logic becomes easy
26      System.out.println("==Starts With==");
27      Z06Analyzer.searchArr(strList01, searchStr,
28        (t, s) -> t.startsWith(s));
```

- The `searchArr` method arguments are:

```
public static void searchArr(String[] strList, String
  searchStr, StringAnalyzer analyzer)
```

Lambda Expressions as Variables

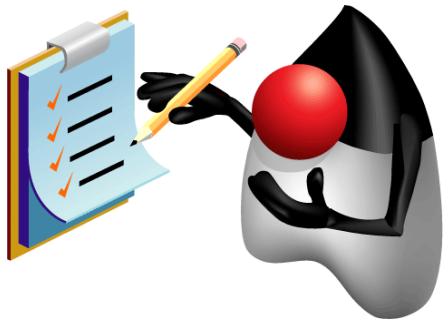
- Lambda expressions can be treated like variables.
- They can be assigned, passed around, and reused.

```
19      // Lambda expressions can be treated like variables
20      StringAnalyzer contains = (t, s) -> t.contains(s);
21      StringAnalyzer startsWith = (t, s) -> t.startsWith(s);
22
23      System.out.println("==Contains==");
24      Z07Analyzer.searchArr(strList, searchStr,
25        contains);
26
27      System.out.println("==Starts With==");
28      Z07Analyzer.searchArr(strList, searchStr,
29        startsWith);
```

Summary

In this lesson, you should have learned how to:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a Lambda Expression



Lab 6-1: Implementing an Interface

This Lab covers the following topics:

- Writing an interface
- Implementing an interface
- Creating references of an interface type
- Casting to interface types

Lab 6-2: Using Java Interfaces

This Lab covers the following topics:

- Updating the banking application to use an interface
- Using interfaces to implement accounts

Lab 6-3: Creating Lambda Expression

This Lab covers the following topics:

- Performing string analysis using lambda expressions
- Practicing writing lambda expressions for the `StringAnalyzer` interface

Quiz

All methods in an interface are:

a. `final`

b. `abstract`

c. `private`

d. `volatile`

Quiz

When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

a. `enums`
b. Executors
c. Functional interfaces
d. Static variables

Quiz

Which is true about the parameters passed into the following lambda expression?

```
(t,s) -> t.contains(s)
```

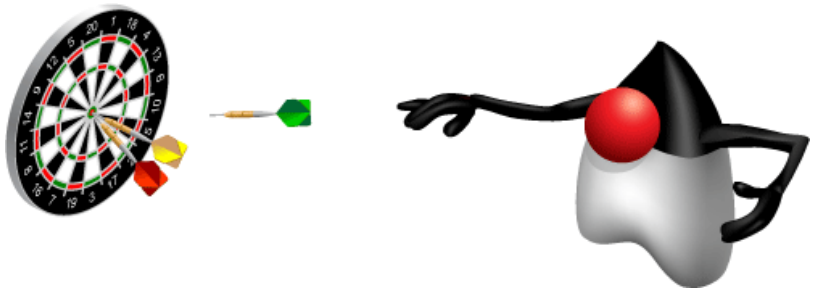
a. Their type is inferred from the context.
b. Their type is executed.
c. Their type must be explicitly defined.
d. Their type is undetermined.

Generics and Collections

Objectives

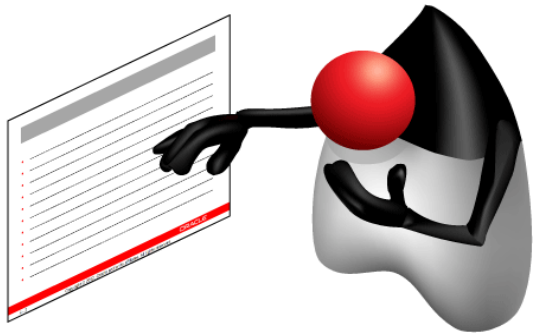
After completing this lesson, you should be able to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `TreeSet`
- Implement a `HashMap`
- Implement a `Deque`
- Order collections



Topics

- Generics
  - Generics with Type Inference Diamond
- Collections
  - Collection Types
  - `List` Interface
  - `ArrayList` Implementation
  - Autoboxing and Unboxing
  - `Set` Interface
  - `Map` Interface
  - `Deque` Interface
  - Ordering Collections
  - `Comparable` Interface
  - `Comparator` Interface



Generics

- Provide flexible type safety to your code
- Move many common errors from run time to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Cache Class Without Generics

```
public class CacheString {
  private String message;
    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```

Generic Cache Class

```
public class CacheAny <T>{

    private T t;

    public void add(T t){
        this.t = t;
    }

    public T get(){
        return this.t;
    }
}
```

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
public static void main(String args[]){
    CacheString myMessage = new CacheString(); // Type
    CacheShirt myShirt = new CacheShirt();     // Type

    //Generics
    CacheAny<String> myGenericMessage = new CacheAny<String>();
    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

    myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

}
```

Generics with Type Inference Diamond

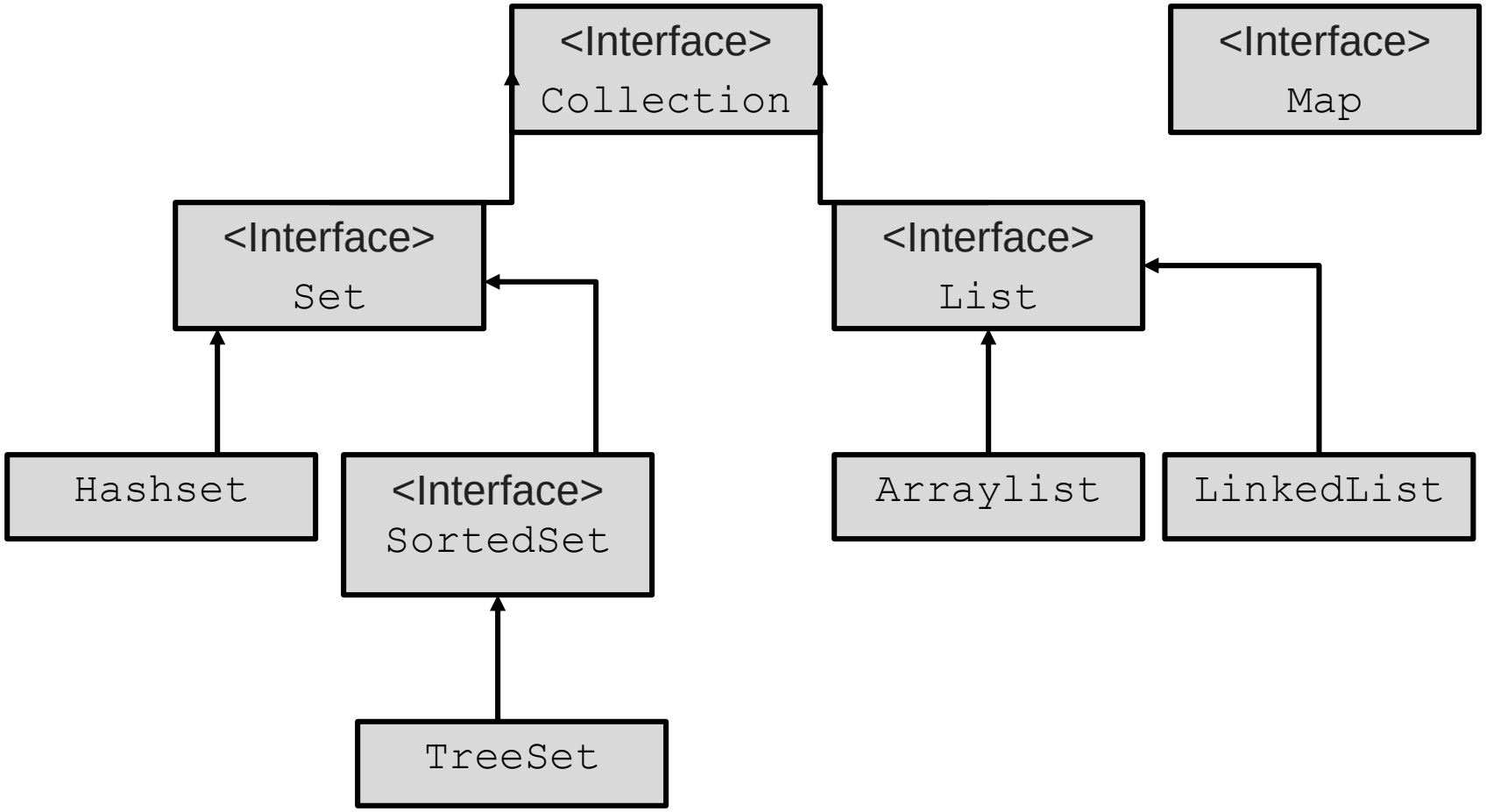
- Syntax:
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

Collections

- A collection is a single object designed to manage a group of objects.
  - Objects in a collection are called *elements.*
  - *Primitives are not allowed in a collection.*
- Various collection types implement many common data structures:
  - Stack, queue, dynamic array, hash
- The Collections API relies heavily on generics for its implementation.

Collection Types



Collection Interfaces and Implementation

Interface	Implementation		
List	ArrayList	LinkedList	
Set	TreeSet	HashSet	LinkedHashSet
Map	HashMap	HashTable	TreeMap
Deque	ArrayDeque		

`List` Interface

- `List` defines generic list behavior.
  - Is an ordered collection of elements
- `List` behaviors include:
  - Adding elements at a specific index
  - Getting an element based on an index
  - Removing an element based on an index
  - Overwriting an element based on an index
  - Getting the size of the list
- `List` allows duplicate elements.

ArrayList

- Is an implementation of the `List` interface
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);
    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add(new Integer(4444)); // ArrayList auto grows
    System.out.println("First Part: " + partList.get(0)); //
First item
    partList.add(0, new Integer(5555)); // Insert an item by
index
```

Autoboxing and Unboxing

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1 public class AutoBox {
2    public static void main(String[] args){
3        Integer intObject = new Integer(1);
4        int intPrimitive = 2;
5
6        Integer tempInteger;
7        int tempPrimitive;
8
9        tempInteger = new Integer(intPrimitive);
10        tempPrimitive = intObject.intValue();
11
12        tempInteger = intPrimitive; // Auto box
13        tempPrimitive = intObject;  // Auto unbox
```

`ArrayList` Without Generics

```
public class OldStyleArrayList {
  public static void main(String args[]){
    List partList = new ArrayList(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Oops a string!");

    Iterator elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = (Integer)(elements.next()); // error?
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

Java example using syntax prior to Java 1.5

Runtime error: `ClassCastException`

Generic `ArrayList`

```
public class GenericArrayList {
  public static void main(String args[]) {
    List<Integer> partList = new ArrayList<>(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Bad Data");  // compiler error now

    Iterator<Integer> elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = elements.next();
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

No cast required.

Generic `ArrayList`: Iteration and Boxing

```
for (Integer partNumberObj:partList){
      int partNumber = partNumberObj; // Demos auto unboxing
      System.out.println("Part number: " + partNumber);
}
```

- The enhanced `for` loop, or `for-each` loop, provides cleaner code.
- No casting is done because of autoboxing and unboxing.

`Set` Interface

- A `Set` is an interface that contains only unique elements.
- A `Set` has no index.
- Duplicate elements are not allowed.
- You can iterate through elements to access them.
- `TreeSet` provides sorted implementation.

`TreeSet`: Implementation of `Set`

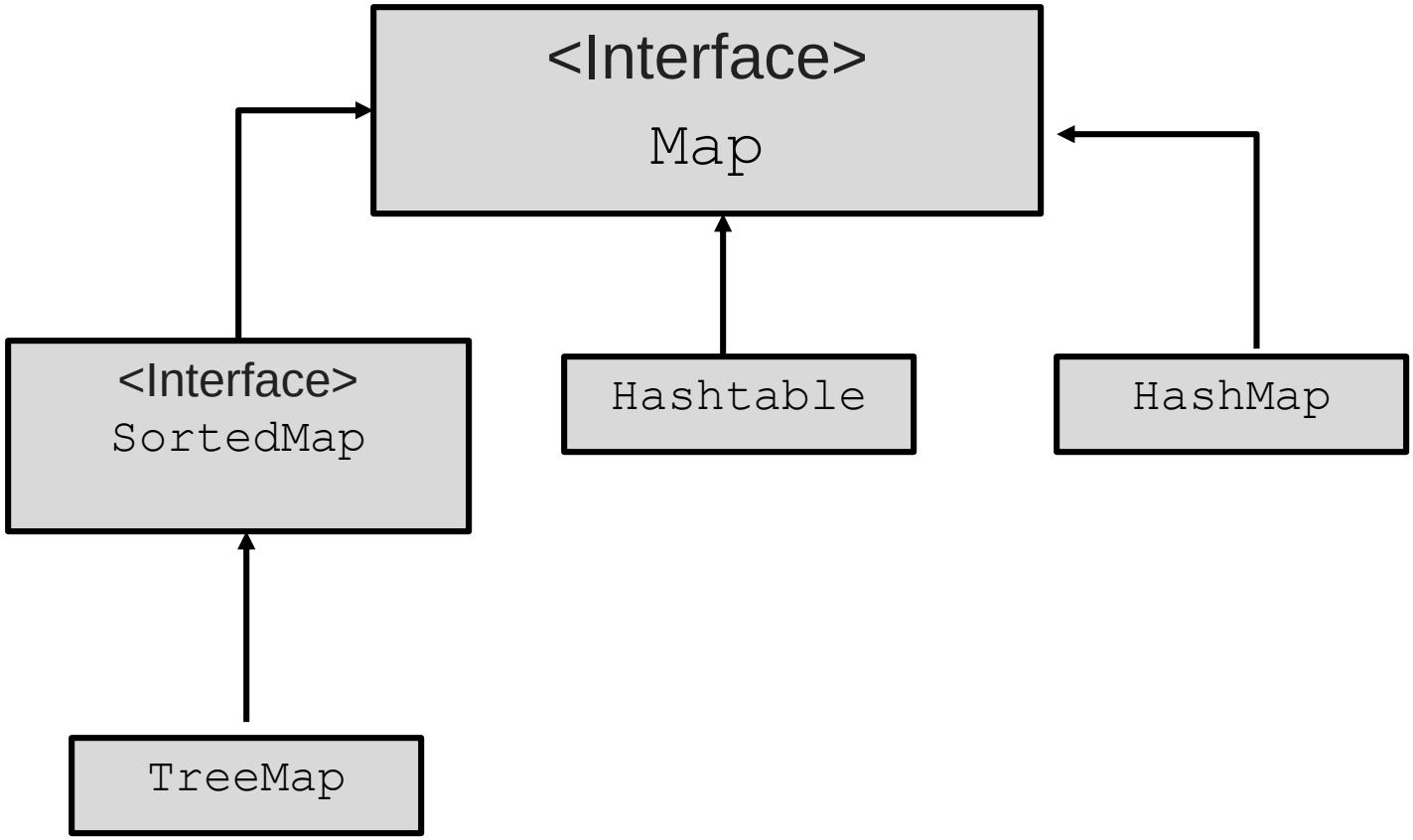
```
public class SetExample {
   public static void main(String[] args){
       Set<String> set = new TreeSet<>();

       set.add("one");
       set.add("two");
       set.add("three");
       set.add("three"); // not added, only unique

       for (String item:set){
           System.out.println("Item: " + item);
       }
   }
}
```

`Map` Interface

- A collection that stores multiple key-value pairs
  - Key: Unique identifier for each element in a collection
  - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

`Map` Types


<Interface>
Map
<Interface>
SortedMap
Hashtable
HashMap
TreeMap

`TreeMap`: Implementation of `Map`

```
public class MapExample {
   public static void main(String[] args){
       Map <String, String> partList = new TreeMap<>();
       partList.put("S001", "Blue Polo Shirt");
       partList.put("S002", "Black Polo Shirt");
       partList.put("H001", "Duke Hat");

       partList.put("S002", "Black T-Shirt"); // Overwrite value
       Set<String> keys = partList.keySet();

       System.out.println("=== Part List ===");
       for (String key:keys){
           System.out.println("Part#: " + key + " " +
                              partList.get(key));
       }
   }
}
```

`Deque` Interface

A collection that can be used as a stack or a queue

- It means a “double-ended queue” (and is pronounced “deck”).
- A queue provides FIFO (first in, first out) operations:
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations:
  - `push(e)` and `pop()` methods

Stack with `Deque`: Example

```
public class TestStack {
    public static void main(String[] args){
        Deque<String> stack = new ArrayDeque<>();
        stack.push("one");
        stack.push("two");
        stack.push("three");

        int size = stack.size() - 1;
        while (size >= 0 ) {
            System.out.println(stack.pop());
            size--;
        }
    }
}
```

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
  - Both are implemented by using generics.
- Using the `Comparable` interface:
  - Overrides the `compareTo` method
  - Provides only one sort option
- The `Comparator` interface:
  - Is implemented by using the `compare` method
  - Enables you to create multiple `Comparator` classes
  - Enables you to create and use numerous sorting options

`Comparable`: Example

```
public class ComparableStudent implements Comparable<ComparableStudent>{
   private String name; private long id = 0; private double gpa = 0.0;

   public ComparableStudent(String name, long id, double gpa){
      // Additional code here
   }
   public String getName(){ return this.name; }
      // Additional code here

   public int compareTo(ComparableStudent s){
       int result = this.name.compareTo(s.getName());
       if (result > 0) { return 1; }
       else if (result < 0){ return -1; }
       else { return 0; }
   }
}
```

`Comparable` Test: Example

```
public class TestComparable {
  public static void main(String[] args){
    Set<ComparableStudent> studentList = new TreeSet<>();

    studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));
    studentList.add(new ComparableStudent("John Adams", 2222, 3.9));
    studentList.add(new ComparableStudent("George Washington", 3333, 3.4));

    for(ComparableStudent student:studentList){
      System.out.println(student);
    }
  }
}
```

`Comparator` Interface

- Is implemented by using the `compare` method
- Enables you to create multiple `Comparator` classes
- Enables you to create and use numerous sorting options

`Comparator`: Example

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0;  // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

Here the compare logic is reversed and results in descending order.

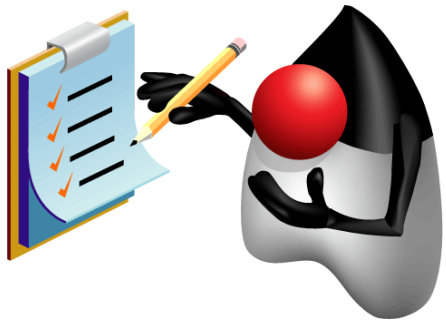
`Comparator` Test: Example

```
1  public class TestComparator {
2    public static void main(String[] args){
3        List<Student> studentList = new ArrayList<>(3);
4        Comparator<Student> sortName = new StudentSortName();
5        Comparator<Student> sortGpa = new StudentSortGpa();
6
7        // Initialize list here
8
9        Collections.sort(studentList, sortName);
10        for(Student student:studentList){
11            System.out.println(student);
12        }
13
14        Collections.sort(studentList, sortGpa);
15        for(Student student:studentList){
16            System.out.println(student);
17        }
18    }
19 }
```

Summary

In this lesson, you should have learned how to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a `Deque`
- Order collections



Lab 7-1 Overview: Counting Part Numbers by Using a `HashMap`

This Lab covers the following topics:

- Creating a map to store a part number and count
- Creating a map to store a part number and description
- Processing the list of parts and producing a report

Lab 7-2 Overview: Implementing Stack by Using a Deque Object

This Lab covers using the `Deque` object to implement a stack.

Quiz

Which of the following is *not* a conventional abbreviation for use with generics?

a. `T`: Table

b. `E`: Element

c. `K`: Key

d. `V`: Value

Quiz

Which interface would you use to create multiple sort options for a collection?

a. `Comparable`

b. `Comparison`

c. `Comparator`

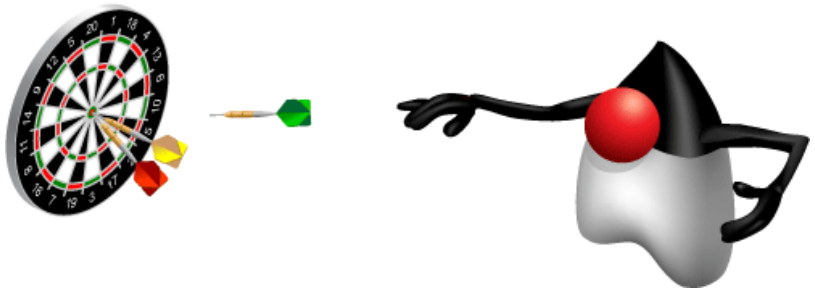
d. `Comparinator`

Collections, Streams, and Filters

Objectives

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods
- Define pipelines in terms of lambdas and collections



Collections, Streams, and Filters

- Iterate through collections using forEach
- Streams and Filters

The Person Class

- Person class
  - Attributes like name, age, address, etc.
- Class created by using the Builder pattern
  - Generates a collection persons for examples
- RoboCall Example
  - An app for contacting people via mail, phone, email
  - Given a list of people query for certain groups
  - Used for test and demo
- Groups queried for
  - Drivers: Persons over the age of 16
  - Draftees: Male persons between 18 and 25 years old
  - Pilots: Persons between 23 and 65 years old

Person Properties

- A Person has the following properties:

```
 9 public class Person {
10   private String givenName;
11   private String surName;
12   private int age;
13   private Gender gender;
14   private String eMail;
15   private String phone;
16   private String address;
17   private String city;
18   private String state;
19   private String code;
```

Builder Pattern

- Allows object creation by using method chaining
  - Easier-to-read code
  - More flexible object creation
  - Object returns itself
  - A fluent approach
- Example

```
260     people.add(
261       new Person.Builder()
262             .givenName("Betty")
263             .surName("Jones")
264             .age(85)
265             .gender(Gender.FEMALE)
266             .email("betty.jones@example.com")
267             .phoneNumber("211-33-1234")
272             .build()
273     );
```

Collection Iteration and Lambdas

- `RoboCall06` Iterating with `forEach`

```
 9 public class RoboCallTest06 {
10
11   public static void main(String[] args){
12
13     List<Person> pl = Person.createShortList();
14
15     System.out.println("\n=== Print List ===");
16     pl.forEach(p -> System.out.println(p));
17
18   }
19 }
```

RoboCallTest07: Stream and Filter

```
10 public class RoboCallTest07 {
11
12   public static void main(String[] args){
13
14     List<Person> pl = Person.createShortList();
15     RoboCall05 robo = new RoboCall05();
16
17     System.out.println("\n=== Calling all Drivers Lambda
===");
18     pl.stream()
19         .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
20         .forEach(p -> robo.roboCall(p));
21
22   }
23 }
```

RobocallTest08: Stream and Filter Again

```
10 public class RoboCallTest08 {
11
12   public static void main(String[] args){
13
14     List<Person> pl = Person.createShortList();
15     RoboCall05 robo = new RoboCall05();
16
17     // Predicates
18     Predicate<Person> allPilots =
19         p -> p.getAge() >= 23 && p.getAge() <= 65;
20
21     System.out.println("\n=== Calling all Drivers Variable
===");
22     pl.stream().filter(allPilots)
23         .forEach(p -> robo.roboCall(p));
24   }
```

`SalesTxn` Class

- Class used in examples and practices to follow
- Stores information about sales transactions
  - Seller and buyer
  - Product quantity and price
- Implemented with a Builder class
- Buyer class
  - Simple class to represent buyers and their volume discount level
- Helper enums
  - BuyerClass: Defines volume discount levels
  - State: Lists the states where transactions take place
  - TaxRate: Lists the sales tax rates for different states

Java Streams

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained
- Method chaining
  - Multiple methods can be called in one statement
- Stream characteristics
  - They are immutable.
  - After the elements are consumed, they are no longer available from the stream.
  - A chain of operations can occur only once on a particular stream (a pipeline).
  - They can be serial (default) or parallel.

The Filter Method

- The Stream class converts collection to a pipeline
  - Immutable data
  - Can only be used once and then tossed
- Filter method uses Predicate lambdas to select items.
- Syntax:

```
15          System.out.println("\n== CA Transations Lambda ==");
16          tList.stream()
17              .filter(t -> t.getState().equals("CA"))
18              .forEach(SalesTxn::printSummary);
```

Method References

In some cases, the lambda expression merely calls a class method.

- `.forEach(t -> t.printSummary())`

- Alternatively, you can use a method reference
  - `.forEach(SalesTxn::printSummary));`
- You can use a method reference in the following situations:
  - Reference to a static method
    - `ContainingClass::staticMethodName`
  - Reference to an instance method
  - Reference to an instance method of an arbitrary object of a particular type (for example, `String::compareToIgnoreCase`)
  - Reference to a constructor
    - `ClassName::new`

Method Chaining

- Pipelines allow method chaining (like a builder).
- Methods include filter and many others.
- For example:

```
21          tList.stream()
22              .filter(t -> t.getState().equals("CA"))
23              .filter(t -> t.getBuyer().getName()
24                  .equals("Acme Electronics"))
25              .forEach(SalesTxn::printSummary);
```

Method Chaining

- You can use compound logical statements.
- You select what is best for the situation.

```
15          System.out.println("\n== CA Transations for ACME ==");
16          tList.stream()
17              .filter(t -> t.getState().equals("CA") &&
18                  t.getBuyer().getName().equals("Acme Electronics"))
19              .forEach(SalesTxn::printSummary);
20
21          tList.stream()
22              .filter(t -> t.getState().equals("CA"))
23              .filter(t -> t.getBuyer().getName()
24                  .equals("Acme Electronics"))
25              .forEach(SalesTxn::printSummary);
```

Pipeline Defined

- A stream pipeline consists of:
  - A source
  - Zero or more intermediate operations
  - One terminal operation
- Examples
  - Source: A Collection (could be a file, a stream, and so on)
  - Intermediate: Filter, Map
  - Terminal: `forEach`

Summary

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods together
- Define pipelines in terms of lambdas and collections

Lab Overview

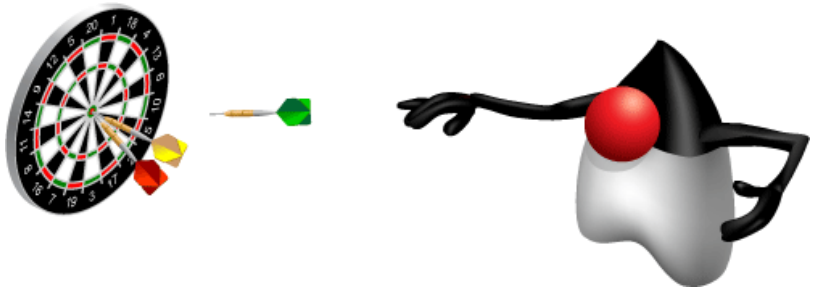
- Lab 8-1: Update RoboCall to use Streams
- Lab 8-2: Mail Sales Executives using Method Chaining
- Lab 8-3: Mail Sales Employees over 50 using Method Chaining
- Lab 8-4: Mail Male Engineering Employees Under 65 Using Method Chaining

Lambda Built-in Functional Interfaces

Objectives

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



Built-in Functional Interfaces

- Lambda expressions rely on functional interfaces
  - Important to understand what an interface does
  - Concepts make using lambdas easier
- Focus on the purpose of main functional interfaces
- Become aware of many primitive variations
- Lambda expressions have properties like those of a variable
  - Use when needed
  - Can be stored and reused

The `java.util.function` Package

- `Predicate`: An expression that returns a `boolean`
- `Consumer`: An expression that performs operations on an object passed as argument and has a void return type
- `Function`: Transforms a T to a U
- `Supplier`: Provides an instance of a T (such as a factory)
- Primitive variations
- Binary variations

Example Assumptions

- The following two declarations are assumed for the examples that follow:

```
14      List<SalesTxn> tList = SalesTxn.createTxnList();
15      SalesTxn first = tList.get(0);
```

Predicate

```
1 package java.util.function;
2
3 public interface Predicate<T> {
4   public boolean test(T t);
5 }
6
```

Predicate: Example

```
16      Predicate<SalesTxn> massSales =
17          t -> t.getState().equals(State.MA);
18
19      System.out.println("\n== Sales - Stream");
20      tList.stream()
21          .filter(massSales)
22          .forEach(t -> t.printSummary());
23
24      System.out.println("\n== Sales - Method Call");
25      for(SalesTxn t:tList){
26          if (massSales.test(t)){
27              t.printSummary();
28          }
29      }
```

Consumer

```
1 package java.util.function;
2
3 public interface Consumer<T> {
4
5     public void accept(T t);
6
7 }
```

Consumer: Example

```
17      Consumer<SalesTxn> buyerConsumer = t ->
18          System.out.println("Id: " + t.getTxnId()
19              + " Buyer: " + t.getBuyer().getName());
20
21      System.out.println("== Buyers - Lambda");
22      tList.stream().forEach(buyerConsumer);
23
24      System.out.println("== First Buyer - Method");
25      buyerConsumer.accept(first);
```

Function

```
1 package java.util.function;
2
3 public interface Function<T,R> {
4
5     public R apply(T t);
6 }
7
```

Function: Example

```
17      Function<SalesTxn, String> buyerFunction =
18          t -> t.getBuyer().getName();
19
20      System.out.println("\n== First Buyer");
21      System.out.println(buyerFunction.apply(first));
22    }
```

Supplier

```
1 package java.util.function;
2
3 public interface Supplier<T> {
4
5     public T get();
6 }
7
```

Supplier: Example

```
15      List<SalesTxn> tList = SalesTxn.createTxnList();
16      Supplier<SalesTxn> txnSupplier =
17          () -> new SalesTxn.Builder()
18              .txnId(101)
19              .salesPerson("John Adams")
20              .buyer(Buyer.getBuyerMap().get("PriceCo"))
21              .product("Widget")
22              .paymentType("Cash")
23              .unitPrice(20)
//... Lines ommited
29              .build();
30
31      tList.add(txnSupplier.get());
32      System.out.println("\n== TList");
33      tList.stream().forEach(SalesTxn::printSummary);
```

Primitive Interface

- Primitive versions of all main interfaces
  - Will see these a lot in method calls
- Return a primitive
  - Example: `ToDoubleFunction`
- Consume a primitive
  - Example: `DoubleFunction`
- Why have these?
  - Avoids auto-boxing and unboxing

Return a Primitive Type

```
1 package java.util.function;
2
3 public interface ToDoubleFunction<T> {
4
5     public double applyAsDouble(T t);
6 }
7
```

Return a Primitive Type: Example

```
18      ToDoubleFunction<SalesTxn> discountFunction =
19          t -> t.getTransactionTotal()
20              * t.getDiscountRate();
21
22      System.out.println("\n== Discount");
23      System.out.println(
24          discountFunction.applyAsDouble(first));
```

Process a Primitive Type

```
1 package java.util.function;
2
3 public interface DoubleFunction<R> {
4
5     public R apply(double value);
6 }
7
```

Process Primitive Type: Example

```
 9      A06DoubleFunction test = new A06DoubleFunction();
10
11      DoubleFunction<String> calc =
12            t -> String.valueOf(t * 3);
13
14      String result = calc.apply(20);
15      System.out.println("New value is: " + result);
```

Binary Types

```
1 package java.util.function;
2
3 public interface BiPredicate<T, U> {
4
5     public boolean test(T t, U u);
6 }
7
```

Binary Type: Example

```
14      List<SalesTxn> tList = SalesTxn.createTxnList();
15      SalesTxn first = tList.get(0);
16      String testState = "CA";
17
18      BiPredicate<SalesTxn,String> stateBiPred =
19        (t, s) -> t.getState().getStr().equals(s);
20
21      System.out.println("\n== First is CA?");
22      System.out.println(
23        stateBiPred.test(first, testState));
```

Unary Operator

```
1 package java.util.function;
2
3 public interface UnaryOperator<T> extends
Function<T,T> {
4     @Override
5     public T apply(T t);
6 }
```

UnaryOperator: Example

- If you need to pass in something and return the same type, use the UnaryOperator interface.

```
17      UnaryOperator<String> unaryStr =
18          s -> s.toUpperCase();
19
20        System.out.println("== Upper Buyer");
21        System.out.println(
22          unaryStr.apply(first.getBuyer().getName()));
```

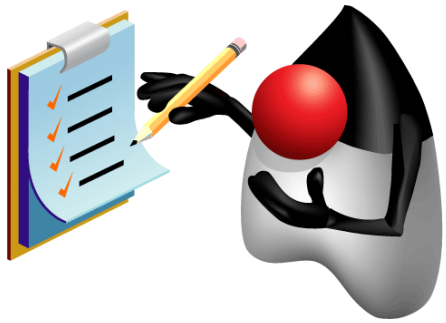
Wildcard Generics Review

- Wildcards for generics are used extensively.
- `? super T`
  - This class and any of its super types
- `? extends T`
  - This class and any of its subtypes

Summary

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



Lab Overview

- Lab 9-1: Create Consumer Lambda Expression
- Lab 9-2: Create a Function Lambda Expression
- Lab 9-3: Create a Supplier Lambda Expression
- Lab 9-4: Create a BiPredicate Lambda Expression

Lambda Operations

Objectives

After completing this lesson, you should be able to:

- Extract data from an object by using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

Streams API

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained
- The Stream class converts collection to a pipeline.
  - Immutable data
  - Can only be used once
  - Method chaining
- Java API doc *is your friend*
- Classes
  - `DoubleStream`, `IntStream`, `LongStream`

Types of Operations

- Intermediate
  - `filter()` `map()` `peek()`
- Terminal
  - `forEach()` `count()` `sum()` `average()` `min()` `max()` `collect()`
- Terminal short-circuit
  - `findFirst()` `findAny()` `anyMatch()` `allMatch()` `noneMatch()`

Extracting Data with Map

`map(Function<? super T,? extends R> mapper)`

- A map takes one `Function` as an argument.
  - A `Function` takes one generic and returns something else.
- Primitive versions of `map`
  - `mapToInt()` `mapToLong()` `mapToDouble()`

Taking a Peek

`peek(Consumer<? super T> action)`

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream.
- Great for printing intermediate results

Search Methods: Overview

- `findFirst()`
  - Returns the first element that meets the specified criteria
- `allMatch()`
  - Returns `true` if all the elements meet the criteria
- `noneMatch()`
  - Returns `true` if none of the elements meet the criteria
- All of the above are short-circuit terminal operations.

Search Methods

- Nondeterministic search methods
  - Used for nondeterministic cases. In effect, situations where parallel is more effective.
  - Results may vary between invocations.
- `findAny()`
  - Returns the first element found that meets the specified criteria
  - Results may vary when performed in parallel.
- `anyMatch()`
  - Returns true if any elements meet the criteria
  - Results may vary when performed in parallel.

Optional Class

- `Optional<T>`
  - A container object that may or may not contain a non-null value
  - If a value is present, `isPresent()` returns true.
  - `get()` returns the value.
  - Found in `java.util`.
- Optional primitives
  - `OptionalDouble OptionalInt OptionalLong`

Lazy Operations

- Lazy operations:
  - Can be optimized
  - Perform only required operations

```
== First CO Bonus ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
```

```
== CO Bonuses ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
  Bonus paid: $7,200.00
Stream start
Executives
CO Executives
  Bonus paid: $6,600.00
Stream start
Executives
CO Executives
  Bonus paid: $8,400.00
```

Stream Data Methods

`count()`

- Returns the count of elements in this stream

`max(Comparator<? super T> comparator)`

- Returns the maximum element of this stream according to the provided `Comparator`

`min(Comparator<? super T> comparator)`

- Returns the minimum element of this stream according to the provided `Comparator`

Performing Calculations

`average()`

- Returns an optional describing the arithmetic mean of elements of this stream
- Returns an empty optional if this stream is empty
- Type returned depends on primitive class.

`sum()`

- Returns the sum of elements in this stream
- Methods are found in primitive streams:
  - `DoubleStream`, `IntStream`, `LongStream`

Sorting

`sorted()`

- Returns a stream consisting of the elements sorted according to natural order

`sorted(Comparator<? super T> comparator)`

- Returns a stream consisting of the elements sorted according to the `Comparator`

Comparator Updates

`comparing(Function<? super T,? extends U> keyExtractor)`

- Allows you to specify any field to sort on based on a method reference or lambda
- Primitive versions of the Function also supported

`thenComparing(Comparator<? super T> other)`

- Specify additional fields for sorting.

`reversed()`

- Reverse the sort order by appending to the method chain.

Saving Data from a Stream

`collect(Collector<? super T,A,R> collector)`

- Allows you to save the result of a stream to a new data structure
- Relies on the `Collectors` class
- Examples
  - `stream().collect(Collectors.toList());`
  - `stream().collect(Collectors.toMap());`

Collectors Class

- **`averagingDouble(ToDoubleFunction<? super T> mapper)`**
  - Produces the arithmetic mean of a double-valued function applied to the input elements
- **`groupingBy(Function<? super T,? extends K> classifier)`**
  - A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map
- **`joining()`**
  - Concatenates the input elements into a String, in encounter order
- **`partitioningBy(Predicate<? super T> predicate)`**
  - Partitions the input elements according to a Predicate

Quick Streams with `Stream.of`

- The `Stream.of` method allows you to easily create a stream.

```
11  public static void main(String[] args) {
12
13    Stream.of("Monday", "Tuesday","Wedensday", "Thursday")
14      .filter(s -> s.startsWith("T"))
15      .forEach(s -> System.out.println("Matching Days: " +
    s));
16  }
```

Flatten Data with `flatMap`

- Use the `flatMap` method to flatten data in a stream.

```
17          Path file = new File("tempest.txt").toPath();
18
19          try{
20
21              long matches = Files.lines(file)
22                  .flatMap(line -> Stream.of(line.split(" ")))
23                  .filter(word -> word.contains("my"))
24                  .peek(s -> System.out.println("Match: " + s))
25                  .count();
26
27              System.out.println("# of Matches: " + matches);
```

Summary

After completing this lesson, you should be able to:

- Extract data from an object using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

Lab Overview

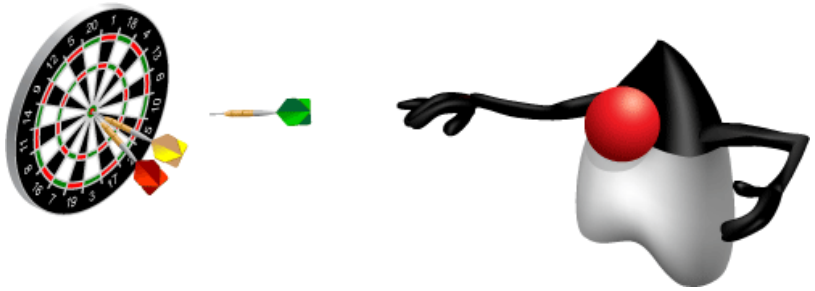
- Lab 10-1: Using Map and Peek
- Lab 10-2: FindFirst and Lazy Operations
- Lab 10-3: Analyze Transactions with Stream Methods
- Lab 10-4: Perform Calculations with Primitive Streams
- Lab 10-5: Sort Transactions with Comparator
- Lab 10-6: Collect Results with Streams
- Lab 10-7: Join Data with Streams
- Lab 10-8: Group Data with Streams

Exceptions and Assertions

Objectives

After completing this lesson, you should be able to:

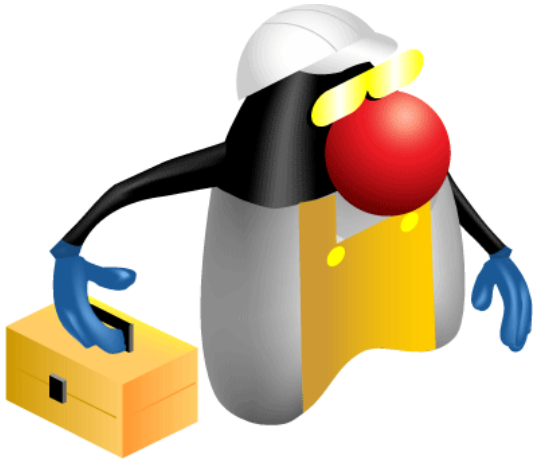
- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions



Error Handling

Applications sometimes encounter errors while executing. Reliable applications should handle errors as gracefully as possible. Errors:

- Should be an exception and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
  - Databases becoming unreachable
  - Hard drives failing



Exception Handling in Java

When you are using Java libraries that rely on external resources, the compiler will require you to “handle or declare” the exceptions that might occur.

- Handling an exception means that you must add in a code block to handle the error.
- Declaring an exception means that you declare that a method may fail to execute successfully.

`try-catch` Statement

The `try-catch` statement is used to handle exceptions.

```
try {
    System.out.println("About to open a file");
    InputStream in =
        new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

This line is skipped if the previous line failed to open the file.

This line runs only if something went wrong in the `try` block.

Exception Objects

A `catch` clause is passed as a reference to a `java.lang.Exception` object.

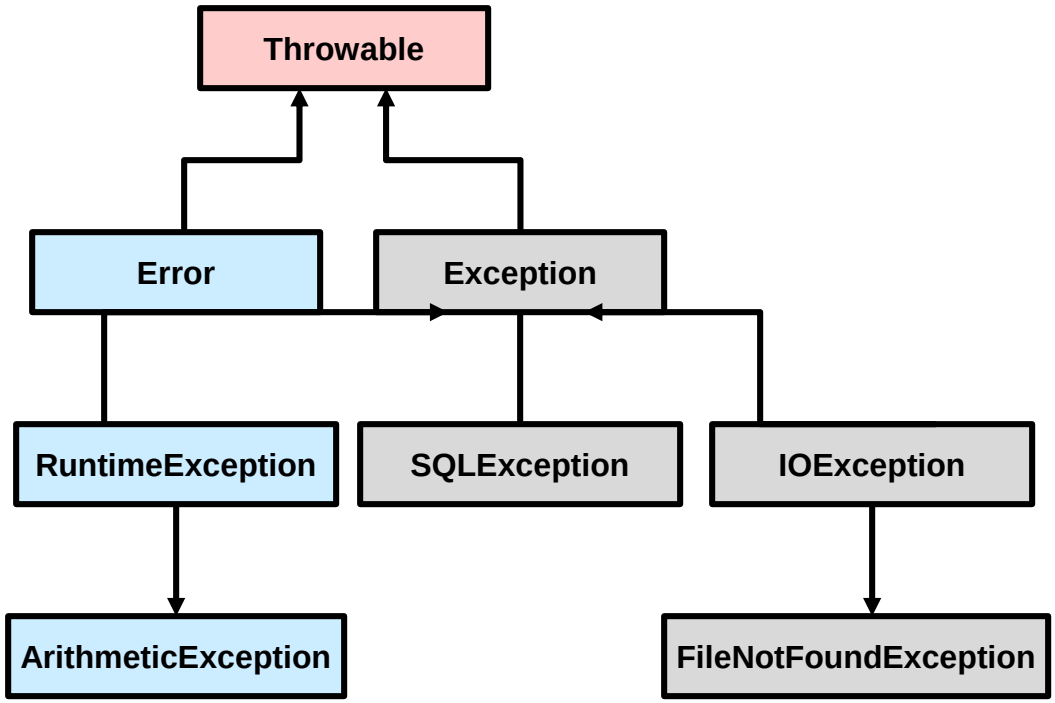
The `java.lang.Throwable` class is the parent class for `Exception` and it outlines several methods that you may use.

```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());

}
```

Exception Categories

The `java.lang.Throwable` class forms the basis of the hierarchy of exception classes. There are two main categories of exceptions:

- Checked exceptions, which must be “handled or declared”
- Unchecked exceptions, which are not typically “handled or declared”



Handling Exceptions

You should always catch the most specific type of exception. Multiple catch blocks can be associated with a single try.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
    in.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
} catch (IOException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
}
```

Order is important. You must catch the most specific exceptions first (that is, child classes before parent classes).

`finally` Clause

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```

A `finally` clause runs regardless of whether or not an `Exception` was generated.

You always want to close open resources.

`try`-with-resources Statement

- The `try`-with-resources statement is a `try` statement that declares one or more resources.
- Any class that implements `java.lang.AutoCloseable` can be used as a resource.

```
System.out.println("About to open a file");
try (InputStream in =
      new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Catching Multiple Exceptions

Using the multi-`catch` clause, a single catch block can handle more than one type of exception.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
     cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Multiple exception types are separated with a vertical bar.

Declaring Exceptions

You may declare that a method throws an exception instead of handling it.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Notice the lack of `catch` clauses. The `try`-with-resources statement is being used only to close resources.

Handling Declared Exceptions

The exceptions that methods may throw must still be handled. Declaring an exception just makes it someone else's job to handle them.

```
public static void main(String[] args) {
   try {
        int data = readByteFromFile();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Method that declared an exception

Throwing Exceptions

The `throw` statement is used to throw an instance of exception.

```
1 import java.io.FileNotFoundException;
2 class DemoThrowsException {
3 public void readFile(String file) throws
4 FileNotFoundException {
5   boolean found = findFile(file);
6   if (!found)
7    throw new FileNotFoundException("Missing file");
8    else {
9           //code to read file
10        }
11    }
12   boolean findFile(String file) {
13        //code to return true if file can be located
14 } }
```

Custom Exceptions

You can create custom exception classes by extending `Exception` or one of its subclasses.

```
class InvalidPasswordException extends Exception {

 InvalidPasswordException() {
    }
 InvalidPasswordException(String message) {
        super(message);
    }
InvalidPasswordException(String message, Throwable cause) {
        super(message, cause);
}
}
```

Assertions

- Use assertions to document and verify the assumptions and internal logic of a single method:
  - Internal invariants
  - Control flow invariants
  - Class invariants
- Inappropriate uses of assertions
  - Do not use assertions to check the parameters of a public method.
  - Do not use methods that can cause side effects in the assertion check.

Assertion Syntax

There are two forms of the `assert` statement:

- **`assert booleanExpression;`**
  - This statement tests the boolean expression.
  - It does nothing if the boolean expression evaluates to `true`.
  - If the boolean expression evaluates to `false`, this statement throws an `AssertionError`.
- **`assert booleanExpression : expression;`**
  - This form acts just like **`assert booleanExpression;`**.
  - In addition, if the boolean expression evaluates to `false`, the second argument is converted to a string and is used as descriptive text in the `AssertionError` message.

Internal Invariants

```
public class Invariant {

    static void checkNum(int num) {
        int x = num;
        if (x > 0) {
            System.out.print( "number is positive" + x);

        } else if (x == 0) {
            System.out.print("number is zero" + x);
        } else {
            assert (x > 0);
        }
    }
    public static void main(String args[]) {

        checkNum(-4);
    }
}
```

Internal Invariant

Control Flow Invariants

```
1 switch (suit) {
2             case Suit.CLUBS: // ...
3             break;
4             case Suit.DIAMONDS: // ...
5             break;
6             case Suit.HEARTS: // ...
7             break;
8             case Suit.SPADES: // ...
9             break;
10    default:
11    assert false : "Unknown playing card suit";
12    break;
13     }
```

Control Flow Invariant

Class Invariants

```
public class PersonClassInvariant {
    String name;
    String ssn;
    int age;

    private void checkAge()
    {
        assert age >= 18 && age < 150;
    }


    public void changeName(String fname)
    {
        checkAge();
        name=fname;
    }
}
```

Class Invariant

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as it would if the check were not there.
- Assertion checks are disabled by default. Enable assertions with either of the following commands:

```
java -enableassertions MyProgram
```

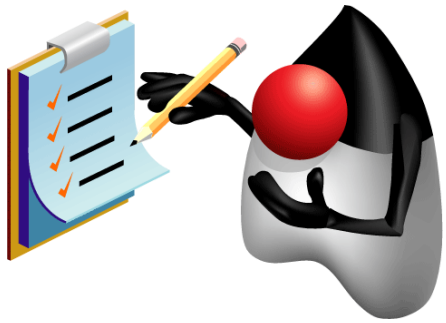
```
java -ea MyProgram
```

- Assertion checking can be controlled on class, package, and package hierarchy basis. See: http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html

Summary

In this lesson, you should have learned how to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions



Lab 11-1 Overview: Catching Exceptions

This Lab covers the following topics:

- Adding `try-catch` statements to a class
- Handling exceptions

Lab 11-2 Overview: Extending `Exception` and Using `throw` and `throws`

This Lab covers the following topics:

- Extending the `Exception` class
- Throwing exceptions using `throw` and `throws`

Quiz

A `NullPointerException` must be caught by using a `try-catch` statement.

a. True
b. False

Quiz

Which of the following types are all checked exceptions (`instanceof`)?

a. `Error`

b. `Throwable`

c. `RuntimeException`

d. `Exception`

Quiz

Which keyword would you use to add a clause to a method stating that the method might produce an exception?

a. `throw`

b. `thrown`

c. `throws`

d. `assert`

Quiz

Assertions should be used to perform user-input validation.

a. True
b. False

Java Date/Time API

Objectives

After completing this lesson, you should be able to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods, and durations
- Apply formatting to local and zoned dates and times

Why Is Date and Time Important?

In the development of applications, programmers often need to represent time and use it to perform calculations:

- The current date and time (locally)
- A date and/or time in the future or past
- The difference between two dates/time in seconds, minutes, hours, days, months, years
- The time or date in another country (time zone)
- The correct time after daylight savings time is applied
- The number of days in the month of February (leap years)
- A time duration (hours, mins, secs) or a period (years, months, days)

Previous Java Date and Time

Disadvantages of `java.util.Date` (`Calendar`, `TimeZone` & `DateFormat`):

- Does not support fluent API approach
- Instances are mutable – not compatible with lambda
- Not thread-safe
- Weakly typed calendars
- One size fits all

Java Date and Time API: Goals

- The classes and methods should be straightforward.
- The API should support a fluent API approach.
- Instances of time/date objects should be immutable. (This is important for lambda operations.)
- Use ISO standards to define date and time.
- Time and date operations should be thread-safe.
- The API should support strong typing, which makes it much easier to develop good code first. (The compiler is your friend!)
- `toString` will always return a human-readable format.
- Allow developers to extend the API easily.

Working with Local Date and Time

The `java.time` API defines two classes for working with local dates and times (without a time zone):

- `LocalDate`:
  - Does not include time
  - A year-month-day representation
  - `toString` – ISO 8601 format (YYYY-MM-DD)
- `LocalTime`:
  - Does not include date
  - Stores hours:minutes:seconds.nanoseconds
  - `toString` – (HH:mm:ss.SSSS)

Working with `LocalDate`

`LocalDate` is a class that holds an event date: a birth date, anniversary, meeting date, and so on.

- A date is a label for a day.
- `LocalDate` uses the ISO calendar by default.
- `LocalDate` does not include time, so it is portable across time zones.
- You can answer the following questions about dates with `LocalDate`:
  - Is it in the future or past?
  - Is it in a leap year?
  - What day of the week is it?
  - What is the day a month from now?
  - What is the date next Tuesday?

`LocalDate`: Example

`next` method

`TUESDAY`

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

  public static void main(String[] args) {
    LocalDate now, bDate, nowPlusMonth, nextTues;
    now = LocalDate.now();
    out.println("Now: " + now);
    bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
    out.println("Java's Bday: " + bDate);
    out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
    out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
    out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
    nowPlusMonth = now.plusMonths(1);
    out.println("The date a month from now: " + nowPlusMonth);
    nextTues = now.with(next(TUESDAY));
    out.println("Next Tuesday's date: " + nextTues);
  }
}
```

`LocalDate` objects are immutable – methods return a new instance.

Working with `LocalTime`

`LocalTime` stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with `LocalTime`
  - When is my lunch time?
  - Is lunch time in the future or past?
  - What is the time 1 hour 15 minutes from now?
  - How many minutes until lunch time?
  - How many hours until bedtime?
  - How do I keep track of just the hours and minutes?

`LocalTime`: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
  public static void main(String[] args) {
    LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
    now = LocalTime.now();
    out.println("The time now is: " + now);
    nowPlus = now.plusHours(1).plusMinutes(15);
    out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
    nowHrsMins = now.truncatedTo(MINUTES);
    out.println("Truncate the current time to minutes: " + nowHrsMins);
    out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
    lunch = LocalTime.of(12, 30);
    out.println("Is lunch in my future? " + lunch.isAfter(now));
    long minsToLunch = now.until(lunch, MINUTES);
    out.println("Minutes til lunch: " + minsToLunch);
    bedtime = LocalTime.of(21, 0);
    long hrsToBedtime = now.until(bedtime, HOURS);
    out.println("How many hours until bedtime? " + hrsToBedtime);
  }
}
```

HOURS, MINUTES

Working with `LocalDateTime`

`LocalDateTime` is a combination of `LocalDate` and `LocalTime`.

- `LocalDateTime` is useful for narrowing events.
- You can answer the following questions with `LocalDateTime`:
  - When is the meeting with corporate?
  - When does my flight leave?
  - When does the course start?
  - If I move the meeting to Friday, what is the date?
  - If the course starts at 9 AM on Monday and ends at 5 PM on Friday, how many hours am I in class?

`LocalTimeDate`: Example

`LocalDateTime`, `LocalDate`, `LocalTime`

`MARCH`

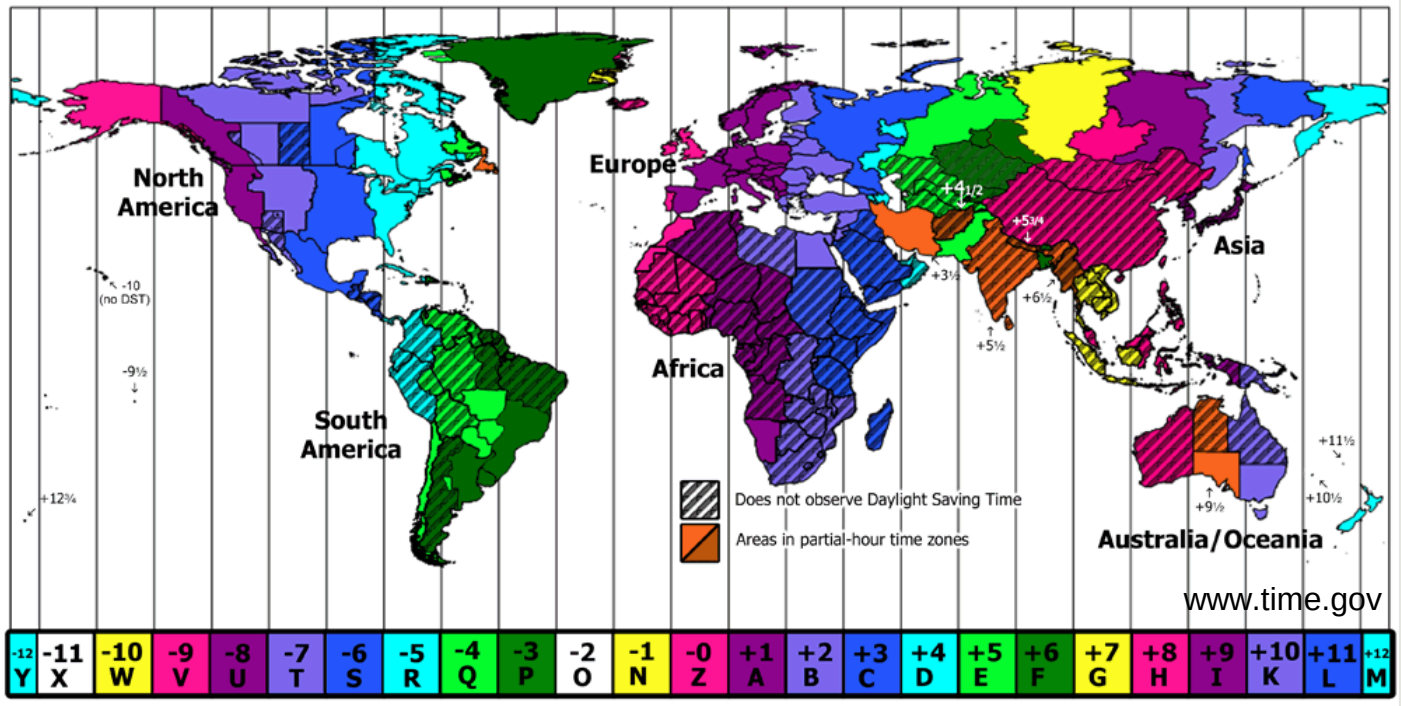
```
import java.time.*;
import static java.time.Month.*;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalDateTimeExample {
  public static void main(String[] args) {
    LocalDateTime meeting, flight, courseStart, courseEnd;
    meeting = LocalDateTime.of(2014, MARCH, 21, 13, 30);
    out.println("Meeting is on: " + meeting);
    LocalDate flightDate = LocalDate.of(2014, MARCH, 31);
    LocalTime flightTime = LocalTime.of(21, 45);
    flight = LocalDateTime.of(flightDate, flightTime);
    out.println("Flight leaves: " + flight);
    courseStart = LocalDateTime.of(2014, MARCH, 24, 9, 00);
    courseEnd = courseStart.plusDays(4).plusHours(8);
    out.println("Course starts: " + courseStart);
    out.println("Course ends:   " + courseEnd);
    long courseHrs = (courseEnd.getHour() - courseStart.getHour()) *
                     (courseStart.until(courseEnd, DAYS) + 1);
    out.println("Course is:     " + courseHrs + " hours long.");
  }
}
```

Combine `LocalDate` and `LocalTime` objects.

Working with Time Zones

Time zones are geographic, but the time in a specific location is defined by the government in that location.

- When a country (and sometimes a state) observes changes (for daylight savings) varies.



Daylight Savings Time Rules

Time changes result in a local hour gap/overlap:

Sunday, March 9, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-5h EST
	1:59:59 AM	UTC-5h EST
	2:00:00 -> 3:00:00	UTC-4h EDT
	3:00:01 AM	UTC-4h EDT

Starting DST causes a one hour gap.

Sunday, November 2, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-4h EST
	1:59:59 AM	UTC-4h EST
	2:00:00 -> 1:00:00	UTC-5h EDT
	1:00:01 AM	UTC-5h EDT

Ending DST causes a one hour overlap.

Modeling Time Zones

- `ZoneId`: Is a specific location or offset relative to UTC

```
ZoneId nyTZ = ZoneId.of("America/New_York");
ZoneId EST = ZoneId.of("US/Eastern");
ZoneId Romeo = ZoneId.of("Europe/London");
```

- `ZoneOffset`: Extends `ZoneId`; specifies the actual time difference from UTC

```
ZoneOffset USEast = ZoneOffset.of("-5");
ZoneOffset Nepal = ZoneOffset.ofHoursMinutes(5, 45);
ZoneId EST = ZoneId.ofOffset("UTC", USEast);
```

- `ZoneRules`: Is the class used to determine offsets

Creating `ZonedDateTime` Objects

- Stores `LocalDateTime`, `ZoneId`, and `ZoneOffset`

```
ZoneId USEast = ZoneId.of("America/New_York");
LocalDate date = LocalDate.of(2014, MARCH, 23);
LocalTime time = LocalTime.of(9, 30);
LocalDateTime dateTime = LocalDateTime.of(date, time);
ZonedDateTime courseStart = ZonedDateTime.of(date, time, USEast);
ZonedDateTime hereNow = ZonedDateTime.now(USEast).truncatedTo(MINUTES);
System.out.println("Here now:          " + hereNow);
System.out.println("Course start:      " + courseStart);
ZonedDateTime newCourseStart = courseStart.plusDays(2).minusMinutes(30);
System.out.println("New Course Start: " + newCourseStart);
```

```
Here now:          2014-02-19 T 17:00 -05:00[America/New_York]
Course start:      2014-03-23 T 09:30 -04:00[America/New_York]
New Course Start: 2014-03-25 T 09:00 -04:00[America/New_York]
```

Space added to make the fields more clear

Working with `ZonedDateTime` Gaps/Overlaps

Given a meeting date the day before daylight savings (2AM on March 9th), what happens if the meeting is moved out by a day?

```
// DST Begins March 9th, 2014
LocalDate meetDate = LocalDate.of(2014, MARCH, 8);
LocalTime meetTime = LocalTime.of(16, 00);
ZonedDateTime meeting = ZonedDateTime.of(meetDate, meetTime, USEast);
System.out.println("meeting time:     " + meeting);
ZonedDateTime newMeeting = meeting.plusDays(1);
System.out.println("new meeting time: " + newMeeting
```

```
meeting time:     2014-03-08 16:00 -05:00[America/New_York]
new meeting time: 2014-03-09 16:00 -04:00[America/New_York]
```

- The local time is not changed, and the offset is managed correctly.

ZoneRules

- Each time zone (`ZoneId`) has a set of rules that are part of the JDK.
- Date or times that land on time changes can be determined by using the rules.

```
// Ask the rules if there was a gap or overlap
ZoneId USEast = ZoneId.of("America/New_York");
LocalDateTime lateNight = LocalDateTime.of(2014, MARCH, 9, 2, 30);
ZoneOffsetTransition zot = USEast.getRules().getTransition(lateNight);
if (zot != null) {
  if (zot.isGap()) System.out.println("gap");
  if (zot.isOverlap()) System.out.println("overlap");
}
```

- Given the code above, what will print?

Working Across Time Zones

The `OffsetDateTime` class stores a `LocalDateTime` and `ZoneOffset`.

- This is useful for determining `ZonedDateTimes` across time zones.

```
LocalDateTime meeting = LocalDateTime.of(2014, JUNE, 13, 12, 30);
ZoneId SanFran = ZoneId.of("America/Los_Angeles");
ZonedDateTime staffCall = ZonedDateTime.of(meeting, SanFran);
OffsetDateTime = staffCall.toOffsetDateTime();
```

- The offset is used to calculate date/time using zone rules:

```
ZoneId London = ZoneId.of("Europe/London");
OffsetDateTime staffCallOffset = staffCall.toOffsetDateTime();
ZonedDateTime staffCallUK = staffCallOffset.atZoneSameInstant(London);
System.out.println("Staff call (Pacific) is at: " + staffCall);
System.out.println("Staff call (UK) is at:      " + staffCallLondon);
```

Date and Time Methods

Prefix	Example	Use
`now`	`today = LocalDate.now()`	Creates an instance using the system clock
`of`	`meet = LocalTime.of(13, 30)`	Creates an instance by using the parameters passed
`get`	`today.get(DAY_OF_WEEK)`	Returns part of the state of the target
`with`	`meet.withHour(12)`	Returns a copy of the target object with one element changed
`plus, minus`	`nextWeek.plusDays(7)` `sooner.minusMinutes(30)`	Returns a copy of the object with the amount added or subtracted
`to`	`meet.toSecondOfDay()`	Converts this object to another type. Here returns `int` seconds.
`at`	`today.atTime(13, 30)`	Combines this object with another; returns a `LocalDateTime` object
`until`	`today.until`	Calculates the amount of time until another date in terms of the unit
`isBefore, isAfter`	`today.isBefore(lastWeek)`	Compares this object with another on the timeline
`isLeapYear`	`today.isLeapYear()`	Checks if this object is a leap year

Date and Time Amounts

- `Instant` – Stores an instant in time on the time-line
  - Useful for: timestamps, e.g. login events
  - Stored as seconds (`long`) and nanoseconds (`int`)
  - Methods used to compare before and after

```
Instant now = Instant.now();
Thread.sleep(0,1); // long milliseconds, int nanoseconds
Instant later = Instant.now();
System.out.println("now is before later? " + now.isBefore(later));
System.out.println("Now:   " + now);
System.out.println("Later: " + later);
```

```
now is before later? true
Now:   2014-02-21 T 16:11:34.788 Z
Later: 2014-02-21 T 16:11:34.789 Z
```

`toString` includes nanoseconds to three digits

Period

`Period` is a class that holds a date-based amount.

- Years, months, and days based on the ISO-8601 calendar
- Plus and minus work with a conceptual day, thus preserving daylight savings changes

```
Period oneDay = Period.ofDays(1);
System.out.println("Period of one day: " + oneDay);
LocalDateTime beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
ZonedDateTime newYorkTime =
  ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(oneDayYear));
```

The time is preserved, because only "days" are added.

```
Period of one day: P1D
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 12:00 -04:00[America/New_York]
```

Duration

`Duration` is a class that stores a time-based amount.

- Time is measured in actual seconds and nanoseconds.
- Days are treated as 24 hours, and daylight savings is ignored.

```
Duration one24hourDay = Duration.ofDays(1);
System.out.println("Duration of one day: " + one24hourDay);
beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
newYorkTime = ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(one24hourDay));
```

The time is not preserved because 24 hours are added.

```
Duration of one day: PT24H
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 13:00 -04:00[America/New_York]
```

Calculating Between Days

`TemporalUnit` is an interface representing a unit of time.

- Implemented by the `enum` class `ChronoUnit`

```
import static java.time.temporal.ChronoUnit.*;

LocalDate christmas = LocalDate.of(2014, DECEMBER, 25);
LocalDate today = LocalDate.now();
long days = DAYS.between(today, christmas);
System.out.println("There are " + days + " shopping days til Christmas");
```

- `Period` also provides a `between` method

```
Period tilXMas = Period.between(today, christmas);
System.out.println("There are " + tilXMas.getMonths() +
                   " months and " + tilXMas.getDays() +
                   " days til Christmas");
```

Making Dates Pretty

`DateTimeFormatter` produces formatted date/times

- Using predefined constants, patterns letters, or a localized style

```
ZonedDateTime now = ZonedDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ISO_ORDINAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofPattern("EEEE, MMMM dd, yyyy G, hh:mm a VV");
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(now.format(formatter));
```

Predefined `DateTimeFormatter` constants

String pattern

Format style

```
2014-02-21
2014-052-05:00
Friday, February 21, 2014 AD, 03:51 PM America/New_York
Feb 21, 2014 3:51:51 PM
```

Year and day of the year

`FormatStyle.MEDIUM`

Using Fluent Notation

One of the goals of JSR-310 was to make the API fluent.

- Examples:

```
 // Not very readable - is this June 11 or November 6th?
LocalDate myBday = LocalDate.of(1970, 6, 11);

// A fluent approach
myBday = Year.of(1970).atMonth(JUNE).atDay(11);

// Schedule a meeting fluently
LocalDateTime meeting = LocalDate.of(2014, MARCH, 25).atTime(12, 30);

// Schedule that meeting using the London timezone
ZonedDateTime meetingUK = meeting.atZone(ZoneId.of("Europe/London"));

// What time is it in San Francisco for that meeting?
ZonedDateTime earlyMeeting =
  meetingUK.withZoneSameInstant(ZoneId.of("America/Los_Angeles"));
```

Summary

In this lesson, you should have learned how to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods and durations
- Apply formatting to local and zoned dates and times

Practices

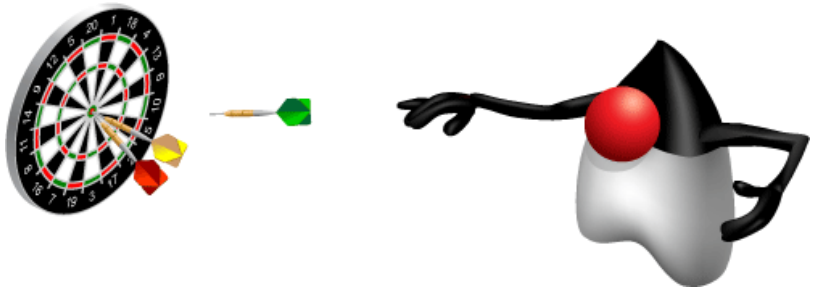
- Lab 12-1: Working with Local Dates and Times
- Lab 12-2: Working with Dates and Times Across Time Zones
- Lab 12-3: Formatting Dates

Java I/O Fundamentals

Objectives

After completing this lesson, you should be able to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use I/O streams to read and write files
- Read and write objects by using serialization



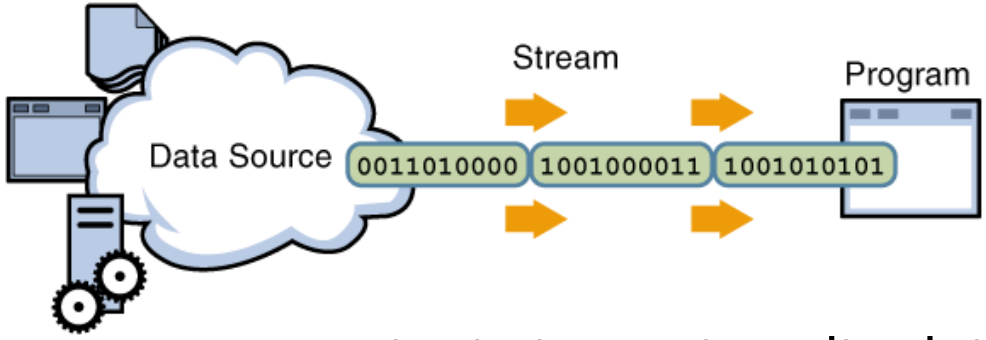
Java I/O Basics

The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

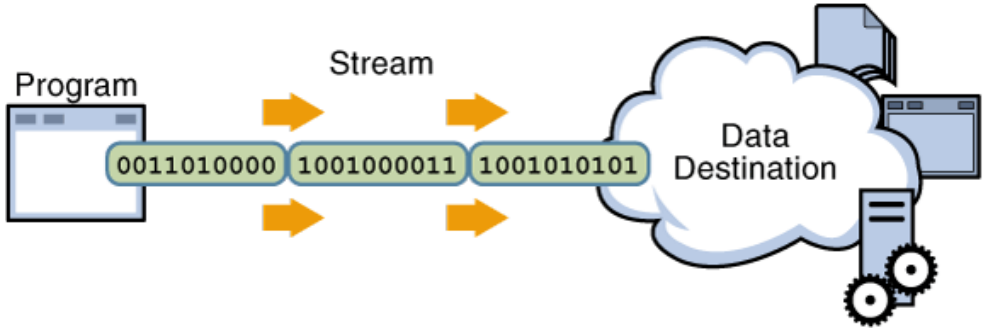
- Java defines an I/O channel as a stream.
- An I/O stream represents an input source or an output destination.
- An I/O stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- I/O streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

I/O Streams

- A program uses an input stream to read data from a source, one item at a time.

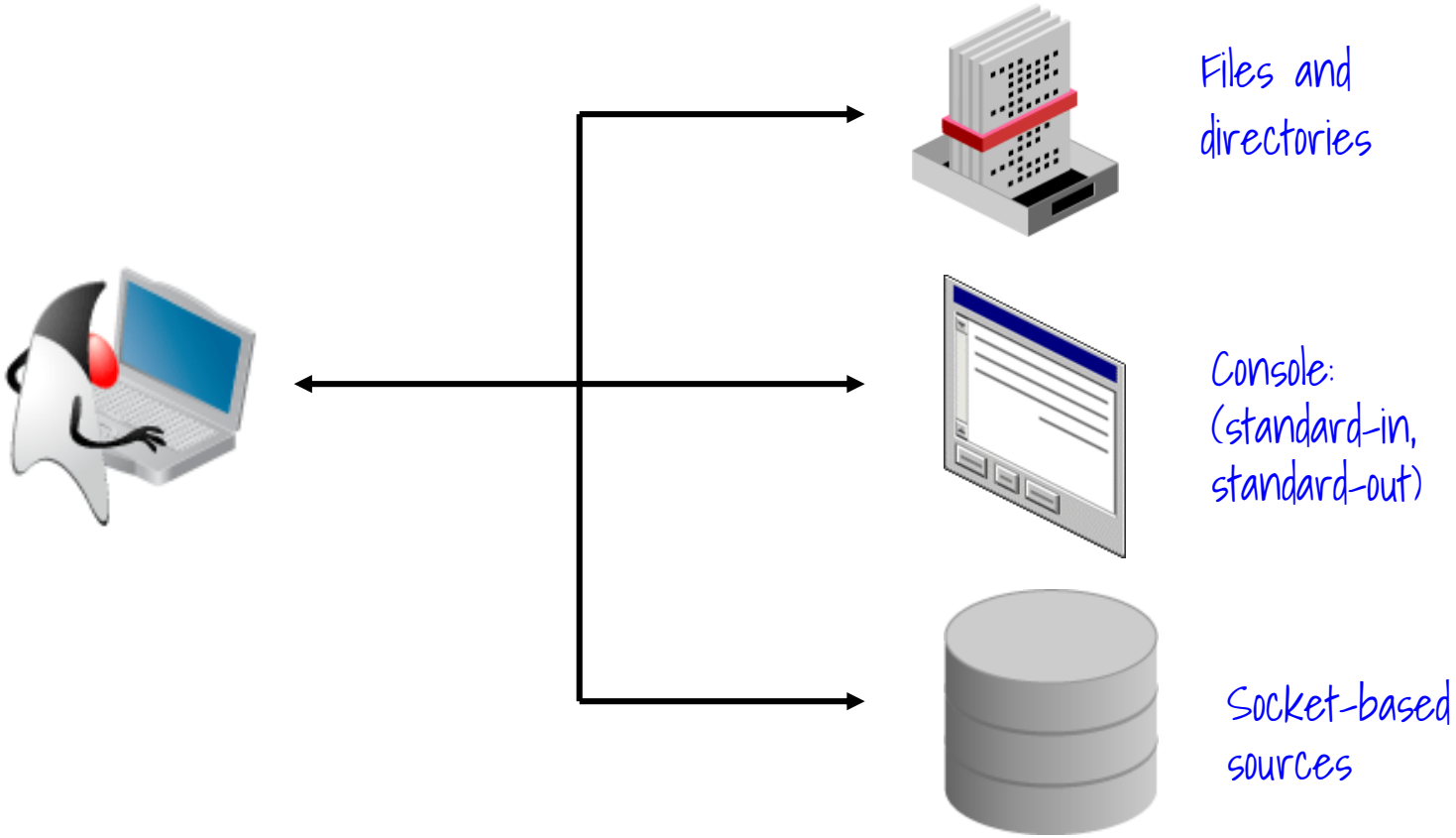


- A program uses an output stream to write data to a destination (sink), one item at time.



I/O Application

Typically, a developer uses input and output in three ways:



Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

Byte Stream `InputStream` Methods

- The three basic `read` methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close();                // Close an open stream
int available();             // Number of bytes available
long skip(long n);           // Discard n bytes from stream
```

Byte Stream `OutputStream` Methods

- The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```

Byte Stream: Example

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteStreamCopyTest {
    public static void main(String[] args) {
        byte[] b = new byte[128];
        // Example use of InputStream methods
        try (FileInputStream fis = new FileInputStream (args[0]);
             FileOutputStream fos = new FileOutputStream (args[1])) {
            System.out.println ("Bytes available: " + fis.available());
           int count = 0; int read = 0;
           while ((read = fis.read(b)) != -1) {
                fos.write(b);
               count += read;
            }
            System.out.println ("Wrote: " + count);
        } catch (FileNotFoundException f) {
            System.out.println ("File not found: " + f);
        } catch (IOException e) {
            System.out.println ("IOException: " + e);
        }
    }
}
```

Note that you must keep track of how many bytes are read into the byte array each time.

Character Stream `Reader` Methods

- The three basic `read` methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

Character Stream `Writer` Methods

- The basic `write` methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

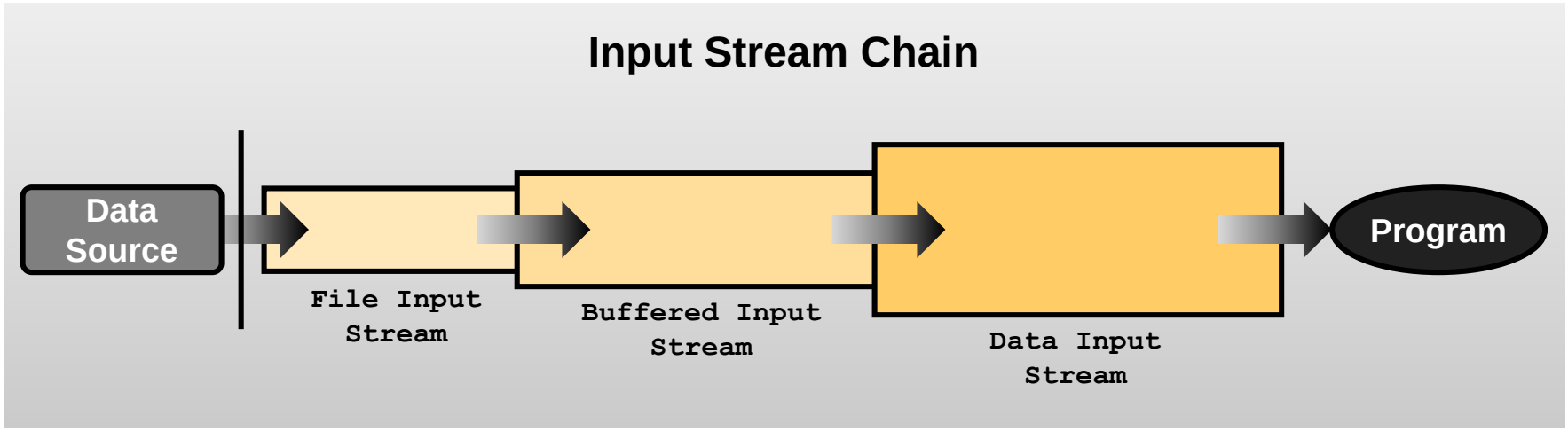
```
void close()
void flush()
```

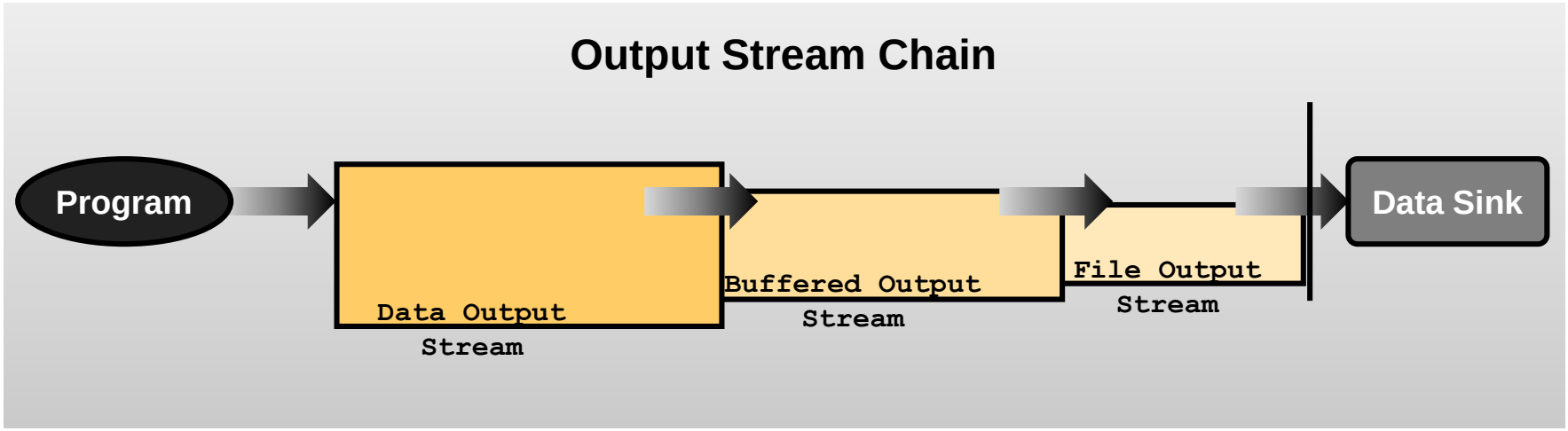
Character Stream: Example

```
import java.io.FileReader; import java.io.FileWriter;
import java.io.IOException; import java.io.FileNotFoundException;

public class CharStreamCopyTest {
    public static void main(String[] args) {
        char[] c = new char[128];
        // Example use of InputStream methods
        try (FileReader fr = new FileReader(args[0]);
             FileWriter fw = new FileWriter(args[1])) {
            int count = 0;
            int read = 0;
            while ((read = fr.read(c)) != -1) {
                 fw.write(c);
                count += read;
            }
            System.out.println("Wrote: " + count + " cha
        } catch (FileNotFoundException f) {
            System.out.println("File " + args[0] + " not found.");
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

Now, rather than a byte array, this version uses a character array.

I/O Stream Chaining





Chained Streams: Example

```
import java.io.BufferedReader; import java.io.BufferedWriter;
import java.io.FileReader; import java.io.FileWriter;
import java.io.FileNotFoundException; import java.io.IOException;

public class BufferedStreamCopyTest {
   public static void main(String[] args) {
       try (BufferedReader bufInput
                = new BufferedReader(new FileReader(args[0]));
            BufferedWriter bufOutput
                = new BufferedWriter(new FileWriter(args[1]))) {
           String line = "";
           while ((line = bufInput.readLine()) != null) {
                bufOutput.write(line);
                bufOutput.newLine();
           }
       } catch (FileNotFoundException f) {
           System.out.println("File not found: " + f);
       } catch (IOException e) {
           System.out.println("Exception: " + e);
       }
   }
}
```

A `FileReader` chained to a `BufferedFileReader`: This allows you to use a method that reads a String.

The character buffer replaced by a `String`. Note that `readLine()` uses the newline character as a terminator. Therefore, you must add that back to the output file.

Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output.*
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input.*
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error.*

Writing to Standard Output

- The `println` and `print` methods are part of the `java.io.PrintStream` class.
- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Reading from Standard Input

```
 7 public class KeyboardInput {
 8
 9     public static void main(String[] args) {
10         String s = "";
11         try (BufferedReader in = new BufferedReader(new
InputStreamReader(System.in))) {
12             System.out.print("Type xyz to exit: ");
13             s = in.readLine();
14             while (s != null) {
15                 System.out.println("Read: " + s.trim());
16                 if (s.equals("xyz")) {
17                     System.exit(0);
18                 }
19                 System.out.print("Type xyz to exit: ");
20                 s = in.readLine();
21             }
22         } catch (IOException e) { // Catch any IO exceptions.
23             System.out.println("Exception: " + e);
24         }
25     }
26 }
```

Chain a buffered reader to an input stream that takes the console input.

Channel I/O

Introduced in JDK 1.4, a channel reads bytes and characters in blocks, rather than one byte or character at a time.

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteChannelCopyTest {
    public static void main(String[] args) {
        try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
             FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
            ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
            fcIn.read(buff);
            buff.position(0);
            fcOut.write(buff);
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

Create a buffer sized the same as the file size, and then read and write the file in a single operation.

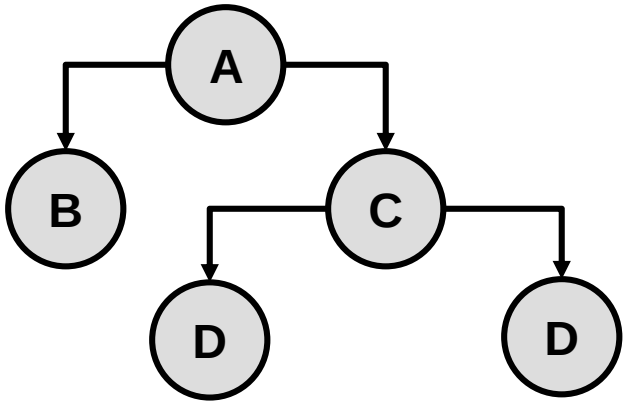
Persistence

Saving data to some type of permanent storage is called persistence. An object that is persistent-capable can be stored on disk (or any other storage device), or sent to another machine to be stored there.

- A non-persisted object exists only as long as the Java Virtual Machine is running.
- Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
- To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

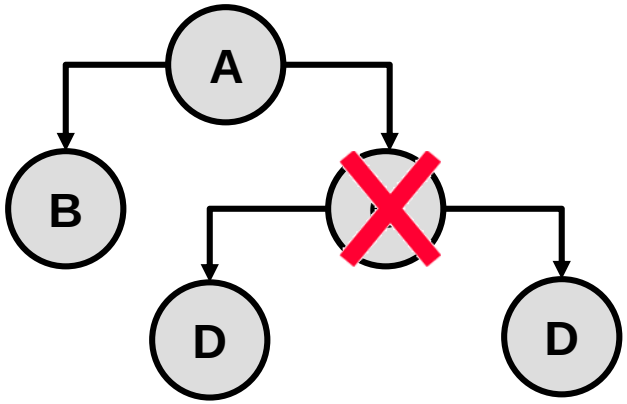
Serialization and Object Graphs

- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object's fields constitutes the *object graph*.



Transient Fields and Objects

- Some object classes are not serializable because they represent transient operating system–specific information.
- If the object graph contains a non-serializable reference, a `NotSerializableException` is thrown and the serialization operation fails.
- Fields that should not be serialized or that do not need to be serialized can be marked with the keyword `transient`.



Transient: Example

```
public class Portfolio implements Serializable {
    public transient FileInputStream inputFile;
    public static int BASE = 100;
    private transient int totalValue = 10;
    protected Stock[] stocks;
}
```

`static` fields are not serialized.

Serialization includes all of the members of the `stocks` array.

- The field access modifier has no effect on the data field being serialized.
- The values stored in static fields are not serialized.
- When an object is deserialized, the values of static fields are set to the values declared in the class. The value of non-static transient fields is set to the default value for the type.

Serial Version UID

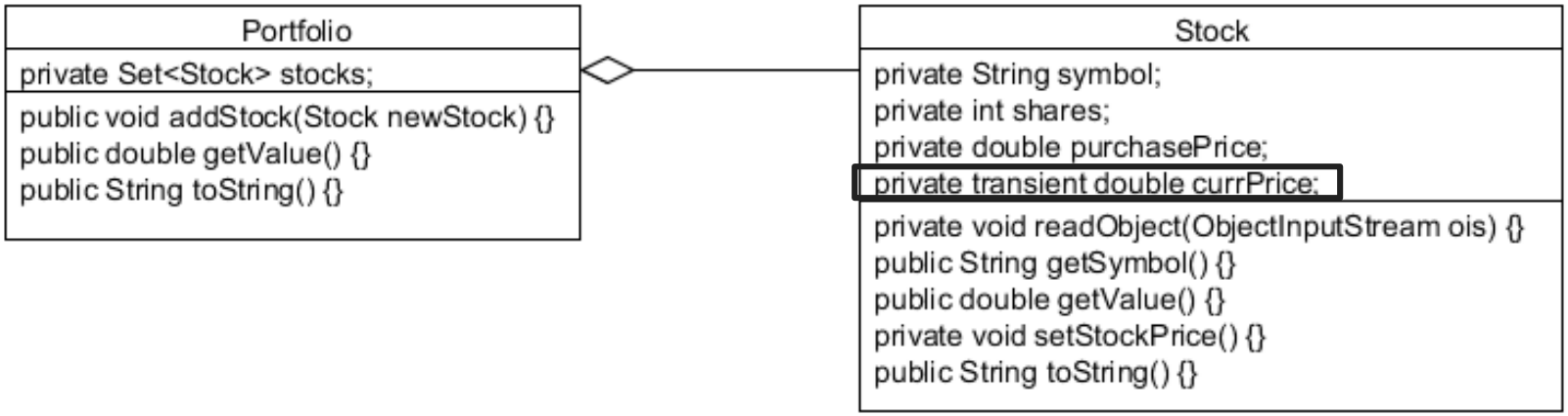
- During serialization, a version number, serialVersionUID, is used to associate the serialized output with the class used in the serialization process.
- After deserialization, the serialVersionUID is checked to verify that the classes loaded are compatible with the object being deserialized.
- If the receiver of a serialized object has loaded classes for that object with different serialVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own serialVersionUID by explicitly declaring a field named `serialVersionUID` as a static final and of type long:

```
private static long serialVersionUID = 42L;
```

Serialization: Example

In this example, a Portfolio is made up of a set of Stocks.

- During serialization, the current price is not serialized, and is, therefore, marked `transient`.
- However, the current value of the stock should be set to the current market price after deserialization.



Writing and Reading an Object Stream

```
public static void main(String[] args) {
    Stock s1 = new Stock("ORCL", 100, 32.50);
    Stock s2 = new Stock("APPL", 100, 245);
    Stock s3 = new Stock("GOOG", 100, 54.67);
    Portfolio p = new   Portfolio(s1, s2, s3);
    try (FileOutputStream fos = new FileOutputStream(args[0]);
         ObjectOutputStream out = new ObjectOutputStream(fos)) {
        out.writeObject(p);
    } catch (IOException i) {
        System.out.println("Exception writing out Portfolio: " + i);
   }
   try (FileInputStream fis = new FileInputStream(args[0]);
        ObjectInputStream in = new ObjectInputStream(fis)) {
       Portfolio newP = (Portfolio)in.readObject();
   } catch (ClassNotFoundException | IOException i) {
       System.out.println("Exception reading in Portfolio: " + i);
}
```

`Portfolio` is the root object.

The `writeObject` method writes the object graph of `p` to the file stream.

The `readObject` method restores the object from the file stream.

Serialization Methods

An object being serialized (and deserialized) can control the serialization of its own fields.

```
public class MyClass implements Serializable {
    // Fields
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        // Write/save additional fields
        oos.writeObject(new java.util.Date());
    }
}
```

`defaultWriteObject` is called to perform the serialization of this class' fields.

- For example, in this class, the current time is written into the object graph.
- During deserialization, a similar method is invoked:

```
private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException {}
```

`readObject`: Example

```
public class Stock implements Serializable {
    private static final long serialVersionUID = 100L;
    private String symbol;
    private int shares;
    private double purchasePrice;
    private transient double currPrice;

    public Stock(String symbol, int shares, double purchasePrice) {
        this.symbol = symbol;
        this.shares = shares;
        this.purchasePrice = purchasePrice;
        setStockPrice();
    }

    // This method is called post-serialization
    private void readObject(ObjectInputStream ois)
                        throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        // perform other initialization
        setStockPrice();
    }
}
```

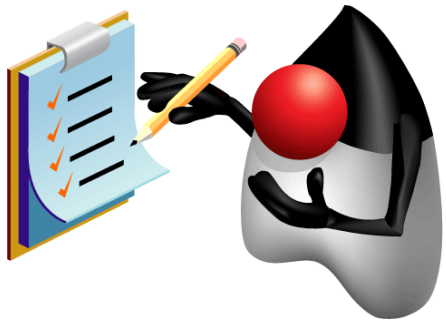
Stock `currPrice` is set by the `setStockPrice` method during creation of the Stock object, but the constructor is not called during deserialization.

Stock `currPrice` is set after the other fields are deserialized.

Summary

In this lesson, you should have learned how to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use streams to read and write files
- Write and read objects by using serialization



Lab 13-1 Overview: Writing a Simple Console I/O Application

This Lab covers the following topics:

- Writing a main class that accepts a file name as an argument
- Using `System` console I/O to read a search string
- Using stream chaining to use the appropriate method to search for the string in the file and report the number of occurrences
- Continuing to read from the console until an exit sequence is entered

Lab 13-2 Overview: Serializing and Deserializing a `ShoppingCart`

This Lab covers the following topics:

- Creating an application that serializes a `ShoppingCart` object that is composed of an `ArrayList` of `Item` objects
- Using the `transient` keyword to prevent the serialization of the `ShoppingCart` total. This will allow items to vary their cost.
- Using the `writeObject` method to store today's date on the serialized stream
- Using the `readObject` method to recalculate the total cost of the cart after deserialization and print the date that the object was serialized

Quiz

The purpose of chaining streams together is to:

a. Allow the streams to add functionality
b. Change the direction of the stream
c. Modify the access of the stream
d. Meet the requirements of JDK 7

Quiz

To prevent the serialization of operating system–specific fields, you should mark the field:

a. `private`

b. `static`

c. `transient`

d. `final`

Quiz

Given the following fragments:

```
public MyClass implements Serializable {
    private String name;
    private static int id = 10;
            private transient String keyword;
    public MyClass(String name, String keyword) {
        this.name = name; this.keyword = keyword;
    }
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Assuming no other changes to the data, what is the value of `name` and `keyword` fields after deserialization of the `mc` object instance?

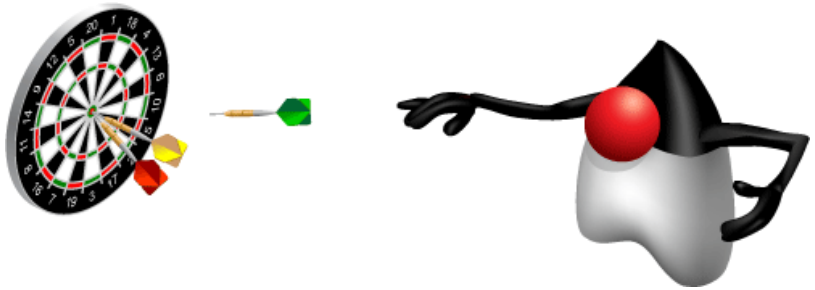
a. `Zim, ""`
b. `Zim, null`
c. `Zim, xyzzy`
d. `"", null`

Java File I/O (NIO.2)

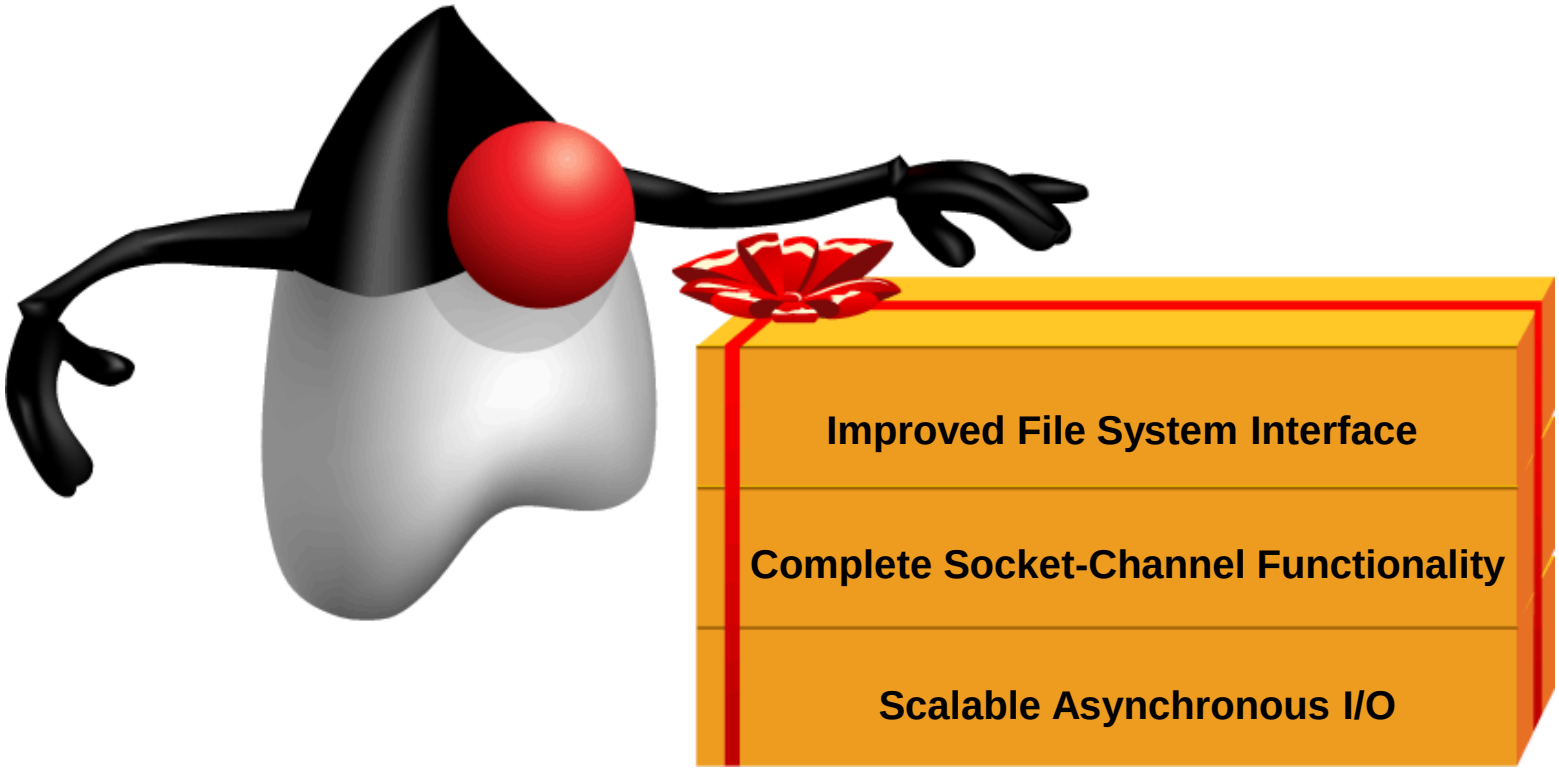
Objectives

After completing this lesson, you should be able to:

- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2

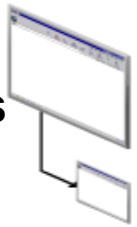


New File I/O API (NIO.2)

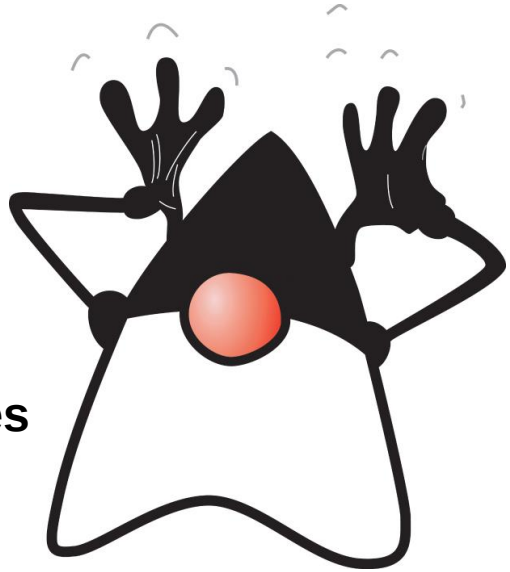


Limitations of `java.io.File`

Does not work well with symbolic links



Scalability issues



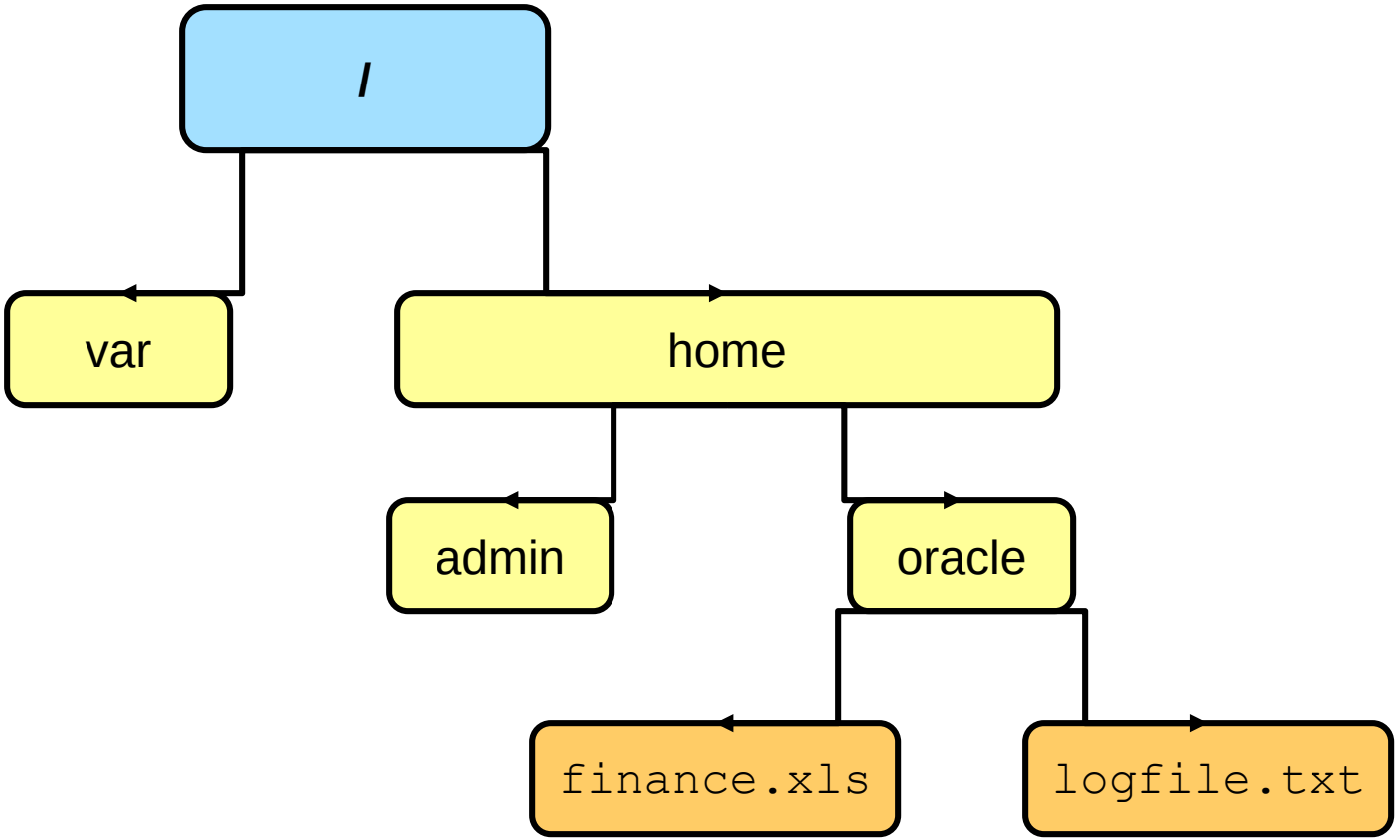
Performance issues

Very limited set of file attributes

Very basic file system access functionality

File Systems, Paths, Files

In NIO.2, both files and directories are represented by a path, which is the relative or absolute location of the file or directory.



Relative Path Versus Absolute Path

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- Example:

```
...
/home/peter/statusReport
...
```

- A relative path must be combined with another path in order to access a file.
- Example:

```
...
clarence/foo
...
```

Java NIO.2 Concepts

Prior to JDK 7, the `java.io.File` class was the entry point for all file and directory operations. With NIO.2, there is a new package and classes:

- `java.nio.file.Path`: Locates a file or a directory by using a system-dependent path
- `java.nio.file.Files`: Using a `Path`, performs operations on files and directories
- `java.nio.file.FileSystem`: Provides an interface to a file system and a factory for creating a `Path` and other objects that access a file system
- All the methods that access the file system throw `IOException` or a subclass.

`Path` Interface

- The `java.nio.file.Path` interface provides the entry point for the NIO.2 file and directory manipulation.

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("/home/oracle/labs/resources/myFile.txt");
```

- To obtain a `Path` object, obtain an instance of the default file system, and then invoke the `getPath` method:

```
Path p1 = Paths.get("/home/oracle/labs/resources/myFile.txt");
Path p2 = Paths.get("/home/oracle", "labs", "resources",
"myFile.txt");
```

`Path` Interface Features

The `Path` interface defines the methods used to locate a file or a directory in a file system. These methods include:

- To access the components of a path:
  - `getFileName`, `getParent`, `getRoot`, `getNameCount`
- To operate on a path:
  - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve`, `relativize`
- To compare paths:
  - `startsWith`, `endsWith`, `equals`

`Path`: Example

```
public class PathTest
    public static void main(String[] args) {
         Path p1 = Paths.get(args[0]);
         System.out.format("getFileName: %s%n", p1.getFileName());
         System.out.format("getParent: %s%n", p1.getParent());
         System.out.format("getNameCount: %d%n", p1.getNameCount());
         System.out.format("getRoot: %s%n", p1.getRoot());
         System.out.format("isAbsolute: %b%n", p1.isAbsolute());
         System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
        System.out.format("toURI: %s%n", p1.toUri());
    }
}
```

```
java PathTest /home/oracle/file1.txt
getFileName: file1.txt
getParent: /home/oracle
getNameCount: 3
getRoot: /
isAbsolute: true
toAbsolutePath: /home/oracle/file1.txt
toURI: file:///home/oracle/file1.txt
```

Removing Redundancies from a `Path`

- Many file systems use "`.`" notation to denote the current directory and "`..`" to denote the parent directory.
- The following examples both include redundancies:

```
/home/./clarence/foo
/home/peter/../clarence/foo
```

- The `normalize` method removes any redundant elements, which includes any "`.`" or "`directory/..`" occurrences.
- Example:

```
Path p = Paths.get("/home/peter/../clarence/foo");
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```

Creating a Subpath

- A portion of a path can be obtained by creating a subpath using the `subpath` method:

```
Path subpath(int beginIndex, int endIndex);
```

- The element returned by `endIndex` is one less that the `endIndex` value.
- Example:

home= 0
oracle = 1
Temp = 2

```
Path p1 = Paths.get ("/home/oracle/Temp/foo/bar");
Path p2 = p1.subpath (1, 3);
```

```
oracle/Temp
```

Include the element at index 2.

Joining Two Paths

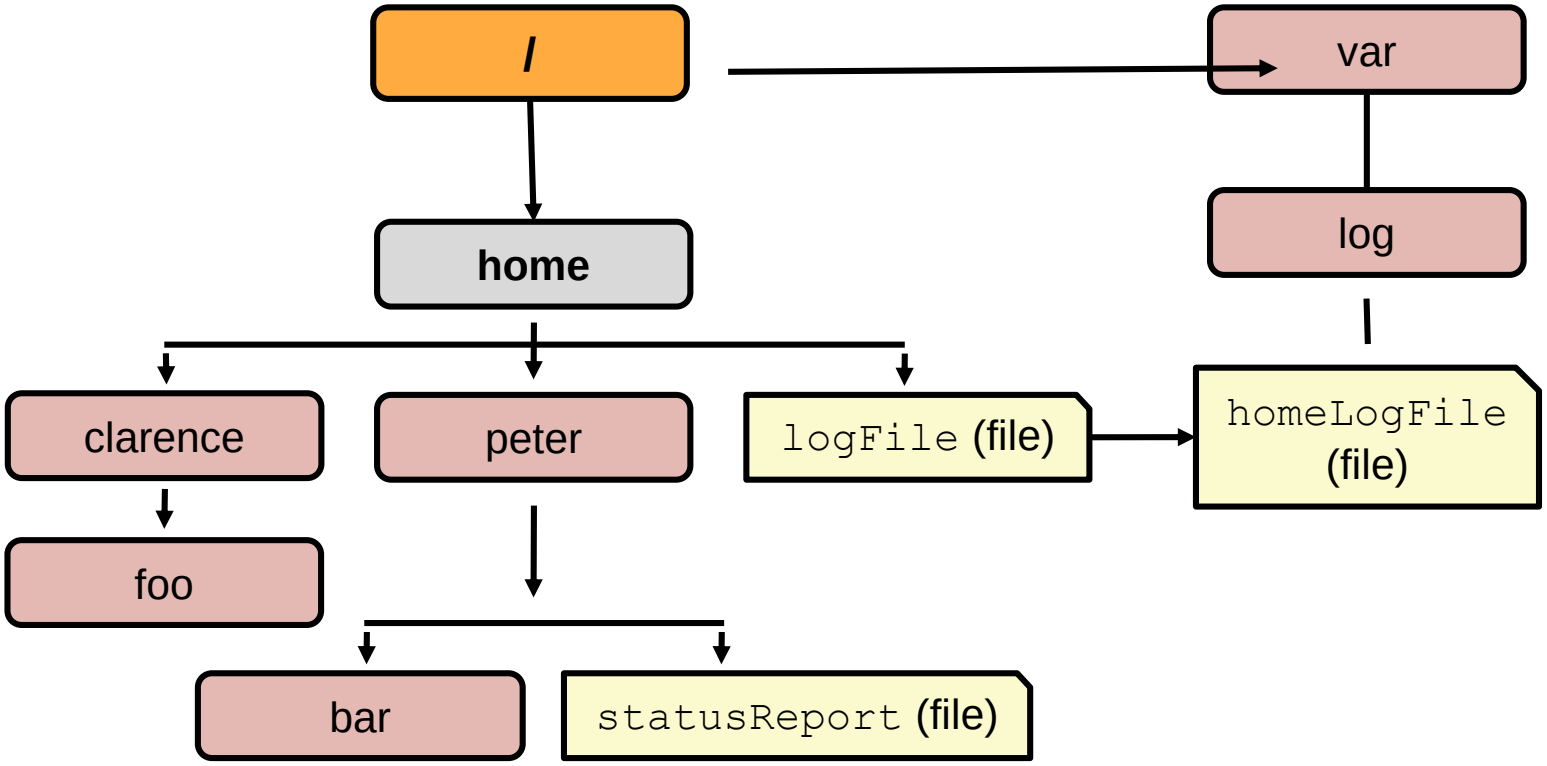
- The `resolve` method is used to combine two paths.
- Example:

```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");     // Returns /home/clarence/foo/bar
```

- Passing an absolute path to the `resolve` method returns the passed-in path.

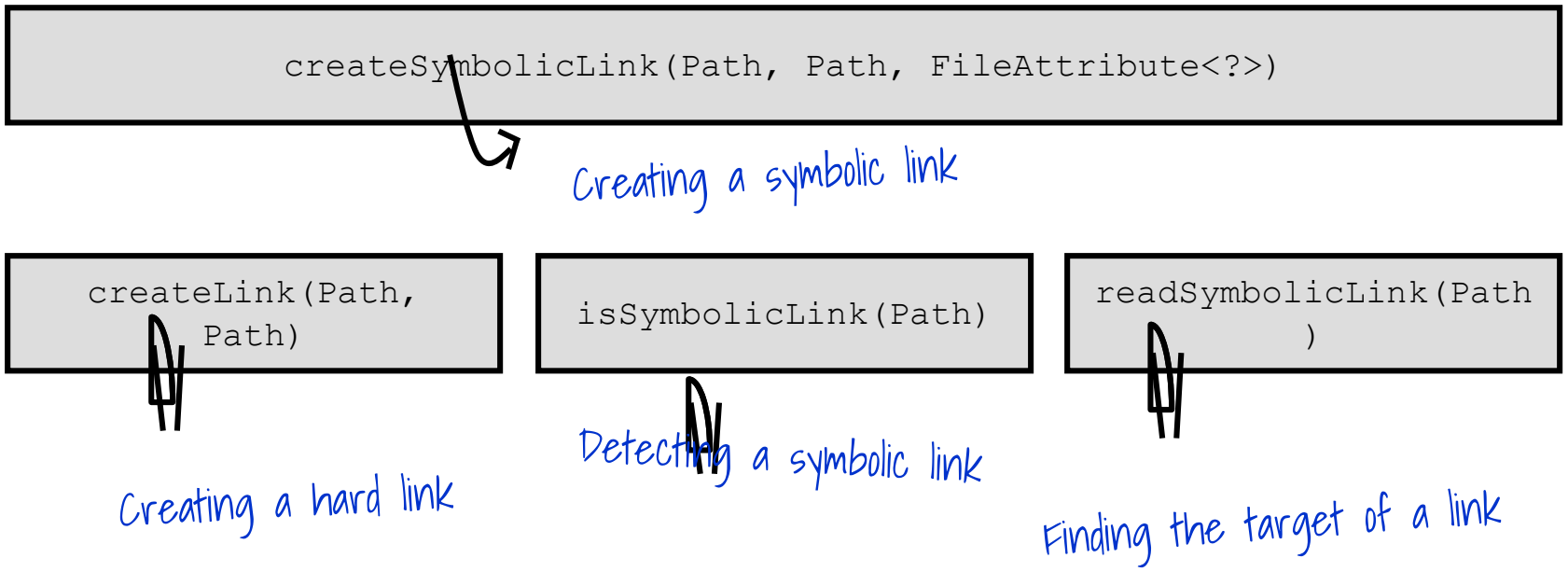
```
Paths.get("foo").resolve("/home/clarence"); // Returns
/home/clarence
```

Symbolic Links



Working with Links

- `Path` interface is "link aware."
- Every `Path` method either:
  - Detects what to do when a symbolic link is encountered, or
  - Provides an option enabling you to configure the behavior when a symbolic link is encountered



`File` Operations

- Checking a File or Directory
- Deleting a File or Directory
- Copying a File or Directory
- Moving a File or Directory
- Managing Metadata
- Reading, Writing, and Creating Files
- Random Access Files
- Creating and Reading Directories

Checking a File or Directory

A `Path` object represents the concept of a file or a directory location. Before you can access a file or directory, you should first access the file system to determine whether it exists using the following `Files` methods:

- `exists(Path p, LinkOption... option)`
  Tests to see whether a file exists. By default, symbolic links are followed.
- `notExists(Path p, LinkOption... option)`
  Tests to see whether a file does not exist. By default, symbolic links are followed.
- Example:

Optional argument

```
Path p = Paths.get(args[0]);
System.out.format("Path %s exists: %b%n", p,
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```

Checking a File or Directory

To verify that a file can be accessed, the `Files` class provides the following `boolean` methods.

- `isReadable(Path)`
- `isWritable(Path)`
- `isExecutable(Path)`

Note that these tests are not atomic with respect to other file system operations. Therefore, the results of these tests may not be reliable once the methods complete.

- The `isSameFile (Path, Path)` method tests to see whether two paths point to the same file. This is particularly useful in file systems that support symbolic links.

Creating Files and Directories

Files and directories can be created using one of the following methods:

```
Files.createFile (Path dir);
Files.createDirectory (Path dir);
```

- The `createDirectories` method can be used to create directories that do not exist, from top to bottom:

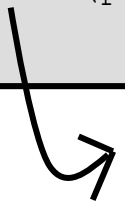
```
Files.createDirectories(Paths.get("/home/oracle/Temp/foo/bar/example"));
```

Deleting a File or Directory

You can delete files, directories, or links. The `Files` class provides two methods:

- `delete(Path)`
- `deleteIfExists(Path)`

```
//...
Files.delete(path);
//...
```



Throws a NoSuchFileException, DirectoryNotEmptyException, or IOException

```
//...
Files.deleteIfExists(Path)
//...
```

No exception thrown

Copying a File or Directory

- You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method.
- When directories are copied, the files inside the directory are not copied.

StandardCopyOption parameters

```
//...
copy(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING
COPY_ATTRIBUTES
NOFOLLOW_LINKS

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

Moving a File or Directory

- You can move a file or directory by using the `move(Path, Path, CopyOption...)` method.
- Moving a directory will not move the contents of the directory.

StandardCopyOption parameters

```
//...
move(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING
ATOMIC_MOVE

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.move(source, target, REPLACE_EXISTING);
```

List the Contents of a Directory

To get a list of the files in the current directory, use the `Files.list()` method.

```
public class FileList {

  public static void main(String[] args) {

    try(Stream<Path> files = Files.list(Paths.get("."))){

      files
          .forEach(line -> System.out.println(line));

    } catch (IOException e){
      System.out.println("Message: " + e.getMessage());
    }
  }
}
```

Walk the Directory Structure

The `Files.walk()` method walks a directory structure.

```
public class A11FileWalk {

  public static void main(String[] args) {

    try(Stream<Path> files = Files.walk(Paths.get("."))){

      files
        .forEach(line -> System.out.println(line));

    } catch (Exception e){
      System.out.println("Message: " + e.getMessage());
    }
  }
}
```

BufferedReader File Stream

The new `lines()` method converts a `BufferedReader` into a stream.

```
public class BufferedRead {
  public static void main(String[] args) {
    try(BufferedReader bReader =
      new BufferedReader(new FileReader("tempest.txt"))){

      bReader.lines()
        .forEach(line ->
            System.out.println("Line: " + line));

    } catch (IOException e){
        System.out.println("Message: " + e.getMessage());
    }
  }
}
```

NIO File Stream

The `lines()` method can be called using NIO classes

```
public class ReadNio {

  public static void main(String[] args) {

    try(Stream<String> lines =
        Files.lines(Paths.get("tempest.txt"))){

      lines.forEach(line ->
        System.out.println("Line: " + line));

    } catch (IOException e){
      System.out.println("Error: " + e.getMessage());
    }
  }
}
```

Read File into `ArrayList`

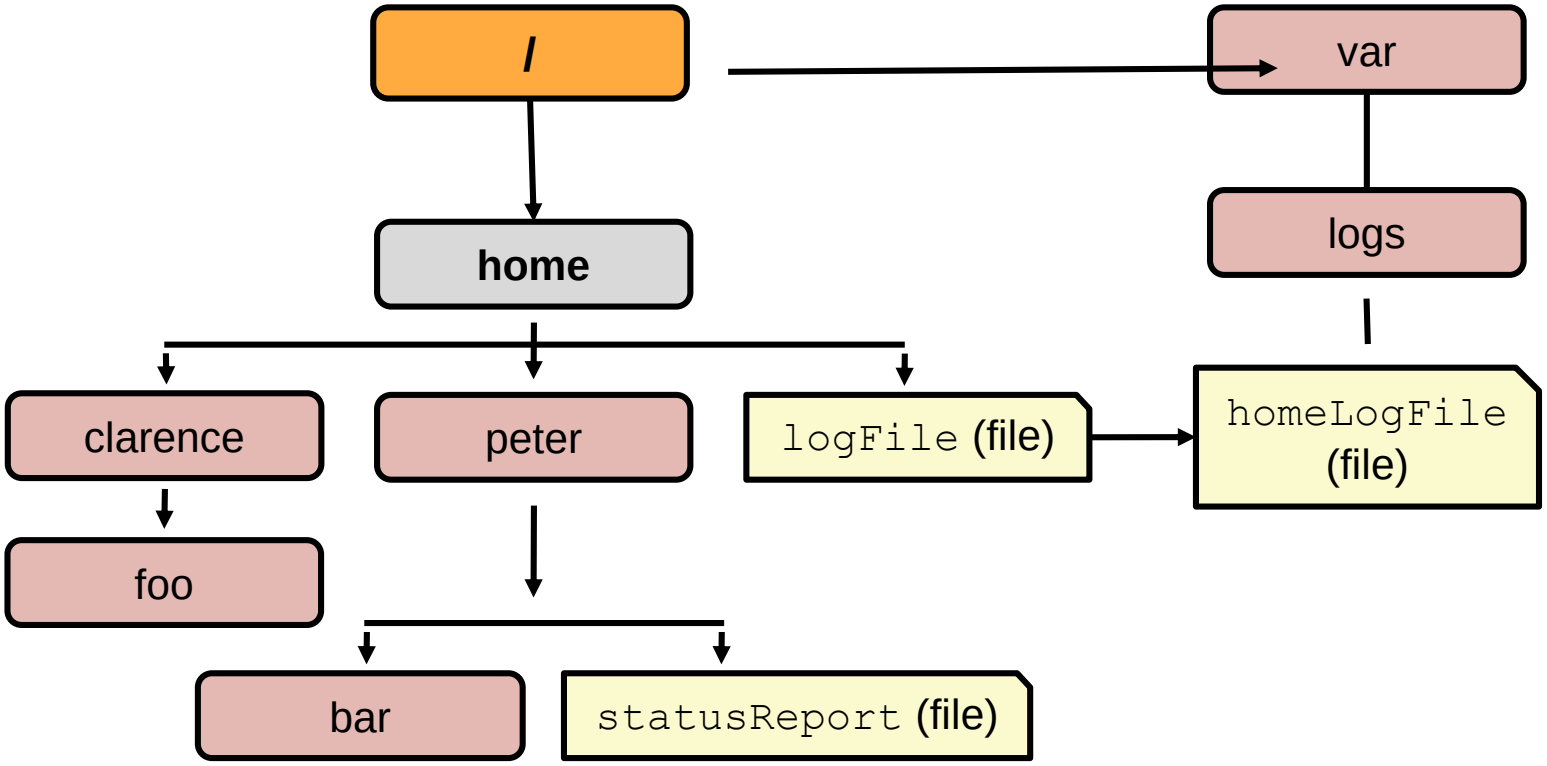
Use `readAllLines()` to load a file into an `ArrayList`.

```
public class ReadAllNio {
  public static void main(String[] args) {
    Path file = Paths.get("tempest.txt");
    List<String> fileArr;

    try{

      fileArr = Files.readAllLines(file);

      fileArr.stream()
        .filter(line -> line.contains("PROSPERO"))
        .forEach(line -> System.out.println(line));

    } catch (IOException e){
      System.out.println("Message: " + e.getMessage());
    }
  }
}
```

Managing Metadata

Method	Explanation
`size`	Returns the size of the specified file in bytes
`isDirectory`	Returns true if the specified `Path` locates a file that is a directory
`isRegularFile`	Returns true if the specified `Path` locates a file that is a regular file
`isSymbolicLink`	Returns true if the specified `Path` locates a file that is a symbolic link
`isHidden`	Returns true if the specified `Path` locates a file that is considered hidden by the file system
`getLastModifiedTime`	Returns or sets the specified file's last modified time
`setLastModifiedTime`	
`getAttribute`	Returns or sets the value of a file attribute
`setAttribute`	

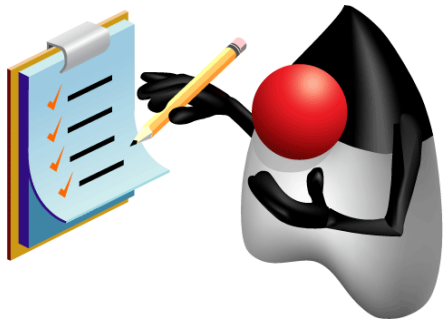
Symbolic Links



Summary

In this lesson, you should have learned how to:

- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2



Lab Overview

Lab 14-1: Working with Files

In this practice, read text files using new features in Java 8 and the lines method.

Lab Overview

Lab 14-2: Working with Directories

In this practice, list directories and files using new features found in Java 8.

Quiz

Given any starting directory path, which `FileVisitor` method(s) would you use to delete a file tree?

a. `preVisitDirectory()`

b. `postVisitDirectory()`

c. `visitFile()`

d. `visitDirectory()`

Quiz

Given a `Path` object with the following path:

`/export/home/duke/../peter/./documents`

What `Path` method would remove the redundant elements?

a. `normalize`
b. `relativize`
c. `resolve`
d. `toAbsolutePath`

Quiz

Given the following fragment:

```
Path p1 = Paths.get("/export/home/peter");
Path p2 = Paths.get("/export/home/peter2");
Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

If the `peter2` directory does not exist, and the `peter` directory is populated with subfolders and files, what is the result?

a. `DirectoryNotEmptyException`
b. `NotDirectoryException`
c. Directory `peter2` is created.
d. Directory `peter` is copied to `peter2`.
e. Directory `peter2` is created and populated with files and directories from `peter`.

Quiz

Given this fragment:

```
Path source = Paths.get(args[0]);
Path target = Paths.get(args[1]);
Files.copy(source, target);
```

Assuming `source` and `target` are not directories, how can you prevent this copy operation from generating `FileAlreadyExistsException`?

a. Delete the `target` file before the copy.
b. Use the `move` method instead.
c. Use the `copyExisting` method instead.
d. Add the `REPLACE_EXISTING` option to the method.

Quiz

To copy, move, or open a file or directory using NIO.2, you must first create an instance of:

a. `Path`

b. `Files`

c. `FileSystem`

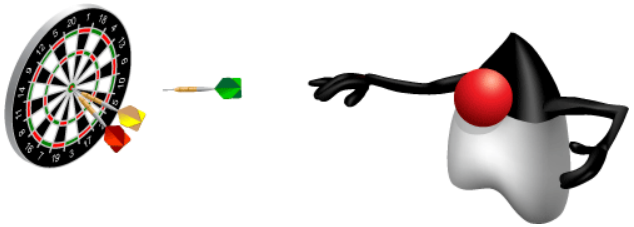
d. `Channel`

Concurrency

Objectives

After completing this lesson, you should be able to:

- Describe operating system task scheduling
- Create worker threads using `Runnable` and `Callable`
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and concurrent atomic to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- **Processes:** A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- **Thread:** A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Legacy `Thread` and `Runnable`

Prior to Java 5, the `Thread` class was used to create and start threads. Code to be executed by a thread is placed in a class, which does either of the following:

- Extends the `Thread` class
  - Simpler code
- Implements the `Runnable` interface
  - More flexible
  - `extends` is still free.

Extending `Thread`

Extend `java.lang.Thread` and override the `run` method:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Implementing `Runnable`

Implement `java.lang.Runnable` and implement the `run` method:

```
public class ExampleRunnable implements Runnable {
    private final String name;

    public ExampleRunnable(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(name + ":" + i);
        }
    }
}
```

The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

Recommended Threading Classes

Traditional `Thread` related APIs are difficult to code properly. Recommended concurrency classes include:

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7

java.util.concurrent.ExecutorService

An `ExecutorService` is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
- Tasks can be:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- Implementing instances can be obtained with `Executors`.

```
ExecutorService es = Executors.newCachedThreadPool();
```

Example `ExecutorService`

This example illustrates using an `ExecutorService` to execute `Runnable` tasks:

```
package com.example;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(new ExampleRunnable("one"));
        es.execute(new ExampleRunnable("two"));
        es.shutdown();
    }
}
```

Execute this Runnable task sometime in the future

Shut down the executor

Shutting Down an `ExecutorService`

Shutting down an `ExecutorService` is important because its threads are nondaemon threads and will keep your JVM from shutting down.

Stop accepting new `Callable`s.

If you want to wait for the `Callable`s to finish

```
es.shutdown();

try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

java.util.concurrent.Callable

The `Callable` interface:

- Defines a task submitted to an `ExecutorService`
- Is similar in nature to `Runnable`, but can:
  - Return a result using generics
  - Throw a checked exception

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

Example `Callable` Task

```
public class ExampleCallable implements Callable {

  private final String name;
  private final int len;
  private int sum = 0;

  public ExampleCallable(String name, int len) {
    this.name = name;
    this.len = len;
  }

  @Override
  public String call() throws Exception {
    for (int i = 0; i < len; i++) {
      System.out.println(name + ":" + i);
      sum += i;
    }
    return "sum: " + sum;
  }
}
```

Return a String from this task: the sum of the series

java.util.concurrent.Future

The `Future` interface is used to obtain the results from a `Callable`'s `V call()` method.

`ExecutorService` controls when the work is done.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
    V result = future.get();
} catch (ExecutionException|InterruptedException ex) {

}
```

Gets the result of the `Callable`'s `call` method (blocks if needed).

If the `Callable` threw an `Exception`

Example

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

  ExecutorService es = Executors.newFixedThreadPool(4);
  Future<String> f1 = es.submit(new ExampleCallable("one",10));
  Future<String> f2 = es.submit(new ExampleCallable("two",20));

  try {
    es.shutdown();
    es.awaitTermination(5, TimeUnit.SECONDS);
    String result1 = f1.get();
    System.out.println("Result of one: " + result1);
    String result2 = f2.get();
    System.out.println("Result of two: " + result2);
  } catch (ExecutionException | InterruptedException ex) {
    System.out.println("Exception: " + ex);
  }

}
```

Wait 5 seconds for the tasks to complete

Get the results of tasks f1 and f2

Threading Concerns

- Thread Safety
  - Classes should continue to behave correctly when accessed from multiple threads.
- Performance: Deadlock and livelock
  - Threads typically interact with other threads. As more threads are introduced into an application, the possibility exists that threads will reach a point where they cannot continue.

Shared Data

Static and instance fields are potentially shared by threads.

```
public class SharedValue {
    private int i;

    // Return a unique value
    public int getNext() {
        return i++;
    }
}
```

Potentially shared variable

Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - "Safely" accessing shared fields is your responsibility.

Two threads accessing an instance of the `SharedValue` class might produce the following:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

Zero produced twice

Out of sequence

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables
- Method parameters
- Exception handler parameters
- Immutable data

Atomic Operations

Atomic operations function as a single operation. A single statement in the Java language is not always atomic.

- `i++;`
  - Creates a temporary copy of the value in `i`
  - Increments the temporary copy
  - Writes the new value back to `i`
- `l = 0xffff_ffff_ffff_ffff;`
  - 64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?

Out-of-Order Execution

- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
  - Code optimization may result in out-of-order operation.
  - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
  - You need a way to state that an action happens before another.
  - You need a way to flush changes to shared variables back to main memory.

The `synchronized` Keyword

The `synchronized` keyword is used to create thread-safe code blocks. A `synchronized` code block:

- Causes a thread to write all of its changes to main memory when the end of the block is reached
- Is used to group blocks of code for exclusive execution
  - Threads block until they can get exclusive access
  - Solves the atomic problem

`synchronized` Methods

```
 3 public class SynchronizedCounter {
 4   private static int i = 0;
 5
 6   public synchronized void increment(){
 7     i++;
 8   }
 9
10   public synchronized void decrement(){
11     i--;
12   }
13
14   public synchronized int getValue(){
15     return i;
16   }
17 }
```

`synchronized` Blocks

```
18    public void run(){
19      for (int i = 0; i < countSize; i++){
20        synchronized(this){
21          count.increment();
22          System.out.println(threadName
23                + " Current Count: " + count.getValue());
24        }
25      }
26    }
```

Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.

```
synchronized ( this ) { }
```

- `synchronized` blocks can be nested.

Threading Performance

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

Performance Issue: Examples

- **Deadlock** results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
        synchronized(obj2) {
        }
}
```

Thread 1 pauses after locking obj1's monitor.

```
synchronized(obj2) {
        synchronized(obj1) {
        }
}
```

Thread 2 pauses after locking obj2's monitor.

- **Starvation** and **Livelock**

`java.util.concurrent` Classes and Packages

The `java.util.concurrent` package contains a number of classes that help with your concurrent applications. Here are just a few examples.

- `java.util.concurrent.atomic` package
  - Lock free thread-safe variables
- `CyclicBarrier`
  - A class that blocks until a specified number of threads are waiting for the thread to complete.
- Concurrency collections

The `java.util.concurrent.atomic` Package

The `java.util.concurrent.atomic` package contains classes that support lock-free thread-safe programming on single variables.

```
 7     public static void main(String[] args) {
 8         AtomicInteger ai = new AtomicInteger(5);
 9         System.out.println("New value: "
10           + ai.incrementAndGet());
11         System.out.println("New value: "
12           + ai.getAndIncrement());
13         System.out.println("New value: "
14           + ai.getAndIncrement());
15
16     }
```

An atomic operation increments value to 6 and returns the value.

java.util.concurrent.CyclicBarrier

The `CyclicBarrier` is an example of the synchronizer category of classes provided by `java.util.concurrent`.

```
10 final CyclicBarrier barrier = new CyclicBarrier(2);
// lines omitted
24     public void run() {
25       try {
26         System.out.println("before await - "
27           + threadCount.incrementAndGet());
28         barrier.await();
29         System.out.println("after await - "
30           + threadCount.get());
31       } catch (BrokenBarrierException|InterruptedException
ex) {
32
33       }
```

Two threads must await before they can unblock.

May not be reached

java.util.concurrent.CyclicBarrier

- If line 18 is uncommented, the program will exit

```
 9 public class CyclicBarrierExample implements Runnable{
10     final CyclicBarrier barrier = new CyclicBarrier(2);
11     AtomicInteger threadCount = new AtomicInteger(0);
12
13
14     public static void main(String[] args) {
15       ExecutorService es = Executors.newFixedThreadPool(4);
16
17       CyclicBarrierExample ex = new CyclicBarrierExample();
18       es.submit(ex);
19       //es.submit(ex);
20
21       es.shutdown();
22     }
```

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as `java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a `Collection` is made thread-safe, this does not make its elements thread-safe.

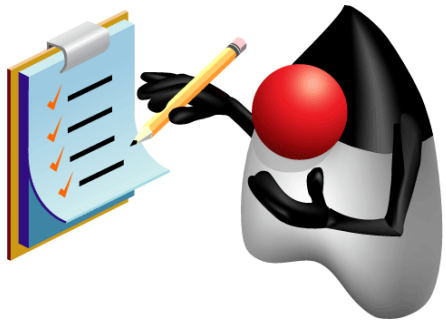
CopyOnWriteArrayList: Example

```
 7 public class ArrayListTest implements Runnable{
 8   private CopyOnWriteArrayList<String> wordList =
 9     new CopyOnWriteArrayList<>();
10
11   public static void main(String[] args) {
12     ExecutorService es = Executors.newCachedThreadPool();
13     ArrayListTest test = new ArrayListTest();
14
15     es.submit(test); es.submit(test);  es.shutdown();
16
17   // Print code here
22   public void run(){
23     wordList.add("A");
24     wordList.add("B");
25     wordList.add("C");
26   }
```

Summary

In this lesson, you should have learned how to:

- Describe operating system task scheduling
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and concurrent atomic to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



Lab 15-1 Overview: Using the `java.util.concurrent` Package

This Lab covers the following topics:

- Using a cached thread pool (`ExecutorService`)
- Implementing `Callable`
- Receiving `Callable` results with a `Future`

Quiz

An `ExecutorService` will always attempt to use all of the available CPUs in a system.

a. True
b. False

Quiz

Variables are thread-safe if they are:

a. `local`

b. `static`

c. `final`

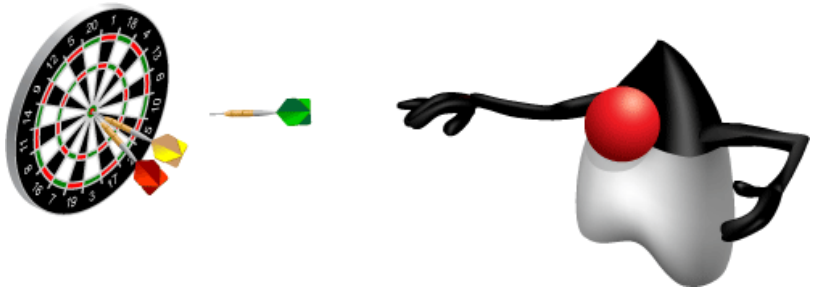
d. `private`

The Fork-Join Framework

Objectives

After completing this lesson, you should be able to:

- Apply the Fork-Join framework



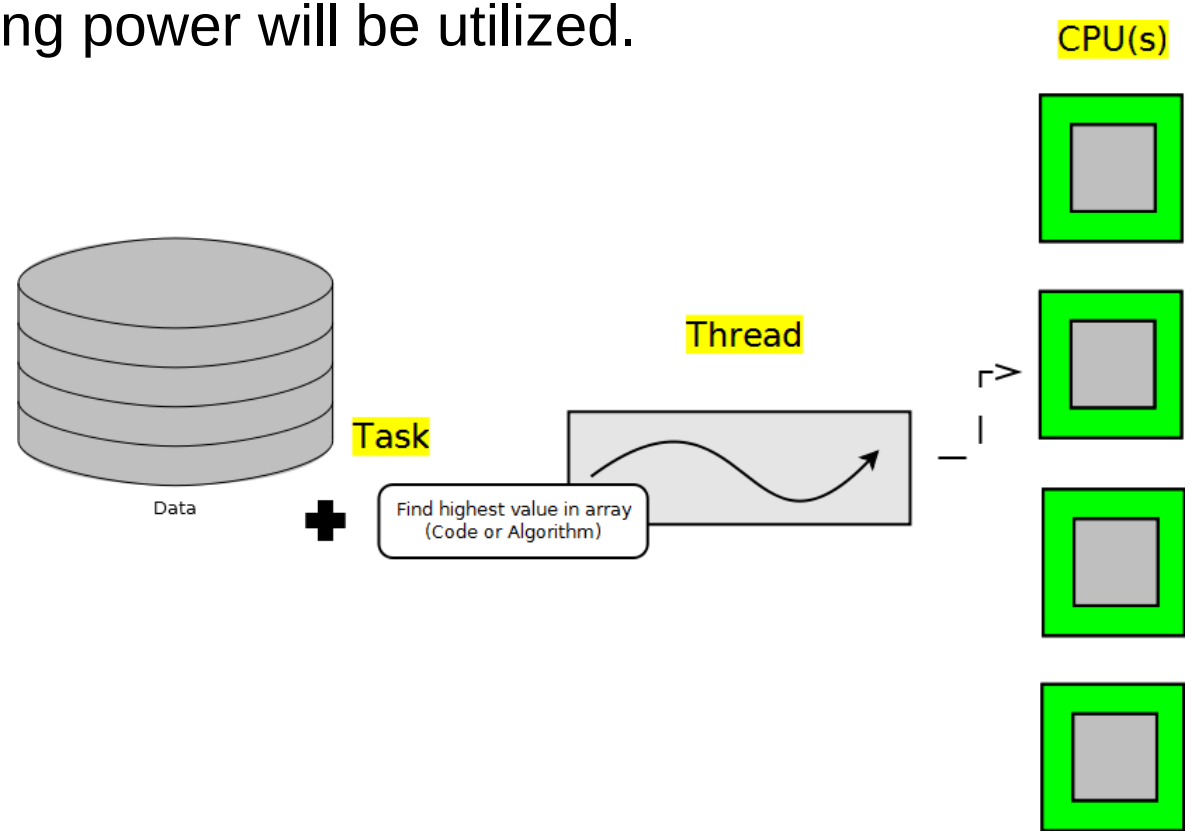
Parallelism

Modern systems contain multiple CPUs. Taking advantage of the processing power in a system requires you to execute tasks in parallel on multiple CPUs.

- Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.
- Some problems can be difficult to execute as parallel tasks.
- Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.
- Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.

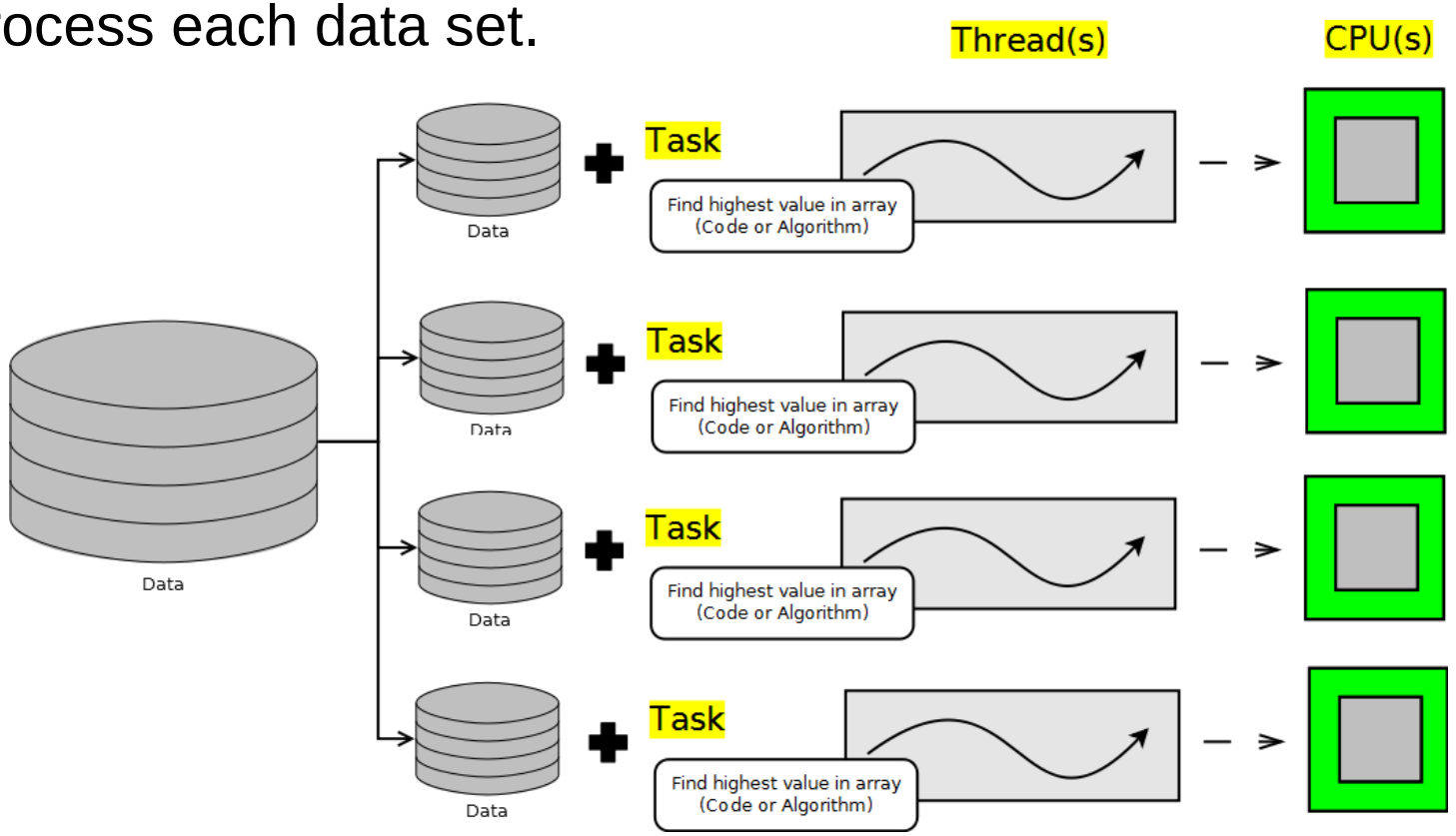
Without Parallelism

Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.



Naive Parallelism

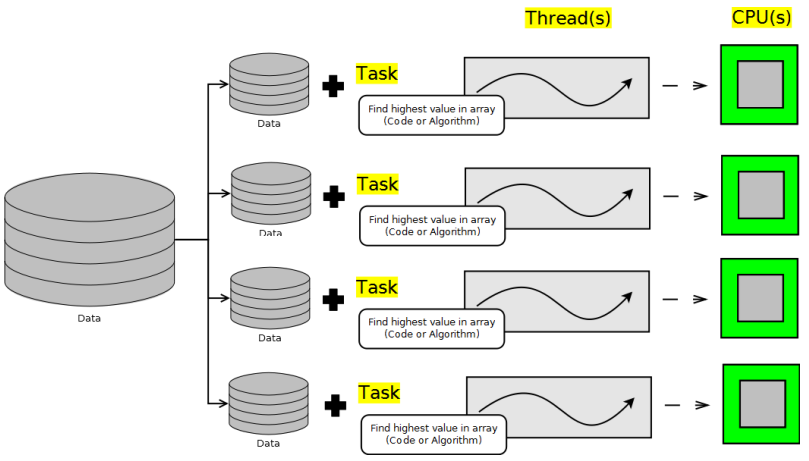
A simple parallel solution breaks the data to be processed into multiple sets: one data set for each CPU and one thread to process each data set.



The Need for the Fork-Join Framework

Splitting datasets into equal sized subsets for each thread to process has a couple of problems. Ideally all CPUs should be fully utilized until the task is finished, but:

- CPUs may run at different speeds
- Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU
- The data being analyzed may require varying amounts of time to process

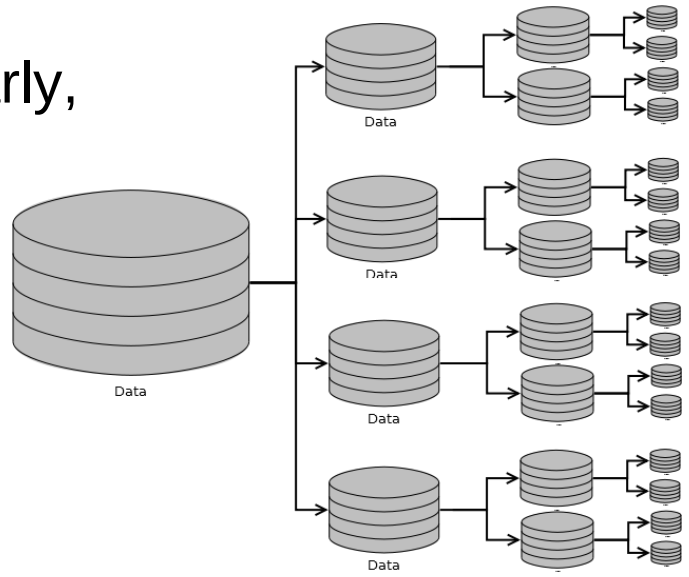


Work-Stealing

To keep multiple threads busy:

- Divide the data to be processed into a large number of subsets
- Assign the data subsets to a thread's processing queue
- Each thread will have many subsets queued

If a thread finishes all its subsets early, it can "steal" subsets from another thread.



A Single-Threaded Example

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
    data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
    if (value > max) {
        max = value;
    }
}
System.out.println("Max value found:" + max);
```

A very large dataset

Fill up the array with values.

Sequentially search the array for the largest value.

java.util.concurrent.ForkJoinTask<V>

A `ForkJoinTask` object represents a task to be executed.

- A task contains the code and data to be processed. Similar to a `Runnable` or `Callable`.
- A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.
  - A `ForkJoinTask` typically creates more `ForkJoinTask` instances until the data to processed has been subdivided adequately.
- Developers typically use the following subclasses:
  - `RecursiveAction`: When a task does not need to return a result
  - `RecursiveTask`: When a task needs to return a result

`RecursiveTask` Example

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end,
int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

Result type of the task

The data to process

Where the work is done.
Notice the generic return type.

`compute` Structure

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

Asynchronously execute

Process in current thread

Block until done

`compute` Example (Below Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
}
```

You decide the threshold.

The range within the array

`compute` Example (Above Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        // find max
    } else {
        int midway = (end - start) / 2 + start;
        FindMaxTask a1 =
    new FindMaxTask(myArray, start, midway, threshold);
        a1.fork();
        FindMaxTask a2 =
    new FindMaxTask(myArray, midway + 1, end, threshold);
        return Math.max(a2.compute(), a1.join());
    }
}
```

Task for left half of data

Task for right half of data

`ForkJoinPool` Example

A `ForkJoinPool` is used to execute a `ForkJoinTask`. It creates a thread for each CPU in the system by default.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

The task's `compute` method is automatically called .

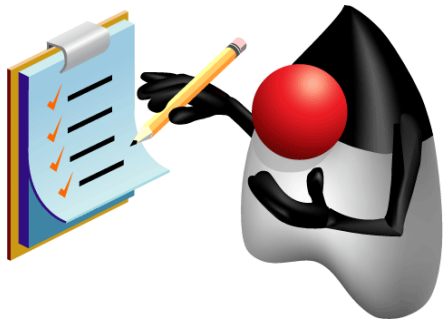
Fork-Join Framework Recommendations

- Avoid I/O or blocking operations.
  - Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.
- Know your hardware.
  - A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.
  - Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.
- Know your problem.
  - Many problems have additional overhead if executed in parallel (parallel sorting, for example).

Summary

In this lesson, you should have learned how to:

- Apply the Fork-Join framework



Lab 16-1 Overview: Using the Fork-Join Framework

This Lab covers the following topics:

- Extending `RecursiveAction`
- Creating and using a `ForkJoinPool`

Quiz

Applying the Fork-Join framework will always result in a performance benefit.

a. True
b. False

Parallel Streams

Objectives

After completing this lesson, you should be able to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

Streams Review

- Pipeline
  - Multiple streams passing data along
  - Operations can be Lazy
  - Intermediate, Terminal, and Short-Circuit Terminal Operations
- Stream characteristics
  - Immutable
  - Once elements are consumed they are no longer available from the stream.
  - Can be sequential (default) or **parallel**

Old Style Collection Processing

```
15          double sum = 0;
16
17          for(Employee e:eList){
18              if(e.getState().equals("CO") &&
19                  e.getRole().equals(Role.EXECUTIVE)){
20                  e.printSummary();
21                  sum += e.getSalary();
22              }
23          }
24
25          System.out.printf("Total CO Executive Pay:
    $%,9.2f %n", sum);
```

New Style Collection Processing

```
15          double result = eList.stream()
16                  .filter(e -> e.getState().equals("CO"))
17                  .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18                  .peek(e -> e.printSummary())
19                  .mapToDouble(e -> e.getSalary())
20                  .sum();
21
22          System.out.printf("Total CO Executive Pay: $%,9.2f
    %n", result);
```

- What are the advantages?
  - Code reads like a problem.
  - Acts on the data set
  - Operations can be lazy.
  - Operations can be serial or parallel.

Stream Pipeline: Another Look

```
13      public static void main(String[] args) {
14
15          List<Employee> eList = Employee.createShortList();
16
17          Stream<Employee> s1 = eList.stream();
18
19          Stream<Employee> s2 = s1.filter(
20              e -> e.getState().equals("CO"));
21
22          Stream<Employee> s3 = s2.filter(
23              e -> e.getRole().equals(Role.EXECUTIVE));
24          Stream<Employee> s4 = s3.peek(e -> e.printSummary());
25          DoubleStream s5 = s4.mapToDouble(e -> e.getSalary());
26          double result = s5.sum();
27
28          System.out.printf("Total CO Executive Pay: $%,9.2f %n",
result);
29      }
```

Styles Compared

Imperative Programming

- Code deals with individual data items.
- Focused on how
- Code does not read like a problem.
- Steps mashed together
- Leaks extraneous details
- Inherently sequential

Streams

- Code deals with data set.
- Focused on what
- Code reads like a problem.
- Well-factored
- No "garbage variables" (Temp variables leaked into scope)
- Code can be sequential or parallel.

Parallel Stream

- May provide better performance
  - Many chips and cores per machine
  - GPUs
- Map/Reduce in the small
- Fork/join is great, but too low level
  - A lot of boilerplate code
  - Stream uses fork/join under the hood
- Many factors affect performance
  - Data size, decomposition, packing, number of cores
- Unfortunately, not a magic bullet
  - Parallel is not always faster

Using Parallel Streams: Collection

- Call from a Collection

```
15          double result = eList.parallelStream()
16                  .filter(e -> e.getState().equals("CO"))
17                  .filter(e ->
e.getRole().equals(Role.EXECUTIVE))
18                  .peek(e -> e.printSummary())
19                  .mapToDouble(e -> e.getSalary())
20                  .sum();
21
22          System.out.printf("Total CO Executive Pay:
$%,9.2f %n", result);
```

Using Parallel Streams: From a Stream

```
27          result = eList.stream()
28                  .filter(e -> e.getState().equals("CO"))
29                  .filter(e -> e.getRole().equals(Role.EXECUTIVE))
30                  .peek(e -> e.printSummary())
31                  .mapToDouble(e -> e.getSalary())
32                  .parallel()
33                  .sum();
34
35          System.out.printf("Total CO Executive Pay: $%,9.2f
%n", result);
```

- Specify with `.parallel` or `.sequential` (default is sequential)
- Choice applies to entire pipeline.
  - Last call wins
- Once again, the API doc is your friend.

Pipelines Fine Print

- Stream pipelines are like Builders.
  - Add a bunch of intermediate operations, and then execute
  - Cannot "branch" or "reuse" pipeline
- Do not modify the source during a query.
- Operation parameters must be stateless.
  - Do not access any state that might change.
  - **This enables correct operation sequentially or in parallel.**
- Best to banish side effects completely.

Embrace Statelessness

```
17  List<Employee> newList02 = new ArrayList<>();
…
23    newList02 = eList.parallelStream() // Good Parallel
24              .filter(e -> e.getDept().equals("Eng"))
25              .collect(Collectors.toList());
```

- Mutate the stateless way
  - The above is preferable.
  - It is designed to parallelize.

Avoid Statefulness

```
15          List<Employee> eList =
Employee.createShortList();
16          List<Employee> newList01 = new ArrayList<>();
17          List<Employee> newList02 = new ArrayList<>();
18
19          eList.parallelStream() // Not Parallel. Bad.
20              .filter(e -> e.getDept().equals("Eng"))
21              .forEach(e -> newList01.add(e));
```

- Temptation is to do the above.
  - **Do not do this. It does not parallelize.**

Streams Are Deterministic for Most Part

```
14          List<Employee> eList = Employee.createShortList();
15
16          double r1 = eList.stream()
17              .filter(e -> e.getState().equals("CO"))
18              .mapToDouble(Employee::getSalary)
19              .sequential().sum();
20
21          double r2 = eList.stream()
22              .filter(e -> e.getState().equals("CO"))
23              .mapToDouble(Employee::getSalary)
24              .parallel().sum();
25
26      System.out.println("The same: " + (r1 == r2));
```

- Will the result be the same?

Some Are Not Deterministic

```
14          List<Employee> eList = Employee.createShortList();
15
16          Optional<Employee> e1 = eList.stream()
17              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
18              .sequential().findAny();
19
20          Optional<Employee> e2 = eList.stream()
21              .filter(e -> e.getRole().equals(Role.EXECUTIVE))
22              .parallel().findAny();
23
24          System.out.println("The same: " +
25              e1.get().getEmail().equals(e2.get().getEmail()));
```

- Will the result be the same?
  - In this case, maybe not.

Reduction

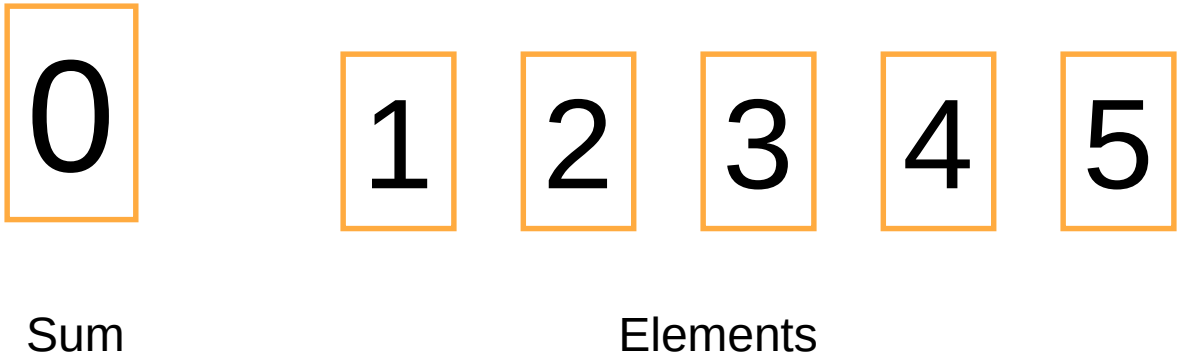
- Reduction
  - An operation that takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
  - Implemented with the `reduce()` method
- Example: `sum` is a reduction with a base value of 0 and a combining function of +.
  - $((((0 + a_1) + a_2) + ...) + a_n)$
  - `.sum()` is equivalent to reduce `(0, (a, b) -> a +b )`
  - `(0, (sum, element) -> sum + element)`

Reduction Fine Print

- If the combining function is associative, reduction parallelizes cleanly
  - Associative means the order does not matter.
  - The result is the same irrespective of the order used to combine elements.
- Examples of: sum, min, max, average, count
  - `.count()` is equivalent to `.map(e -> 1).sum()`.
- **Warning:** If you pass a nonassociative function to `reduce`, you will get the wrong answer. The function must be associative.

Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

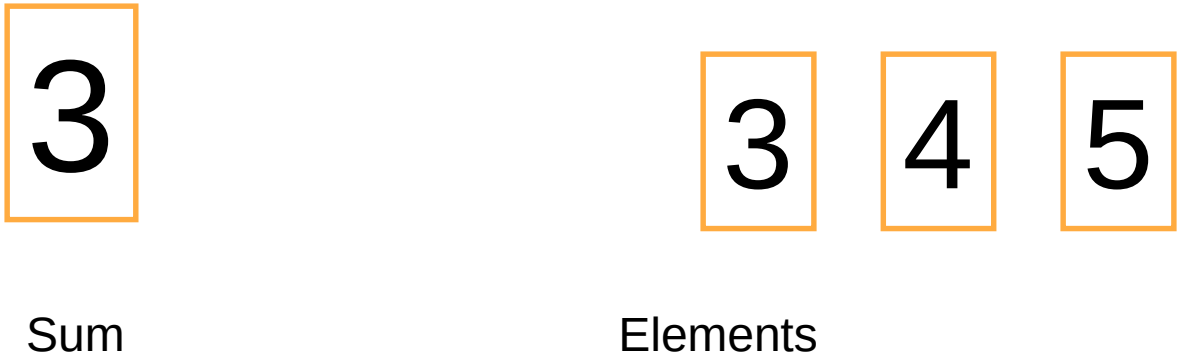


Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

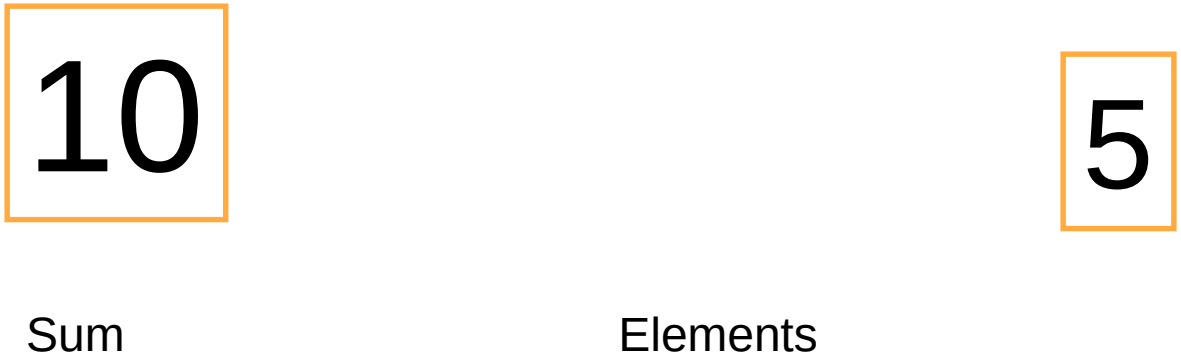


Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```



Reduction: Example

```
18          int r2 = IntStream.rangeClosed(1, 5).parallel()
19              .reduce(0, (sum, element) -> sum + element);
20
21          System.out.println("Result: " + r2);
```

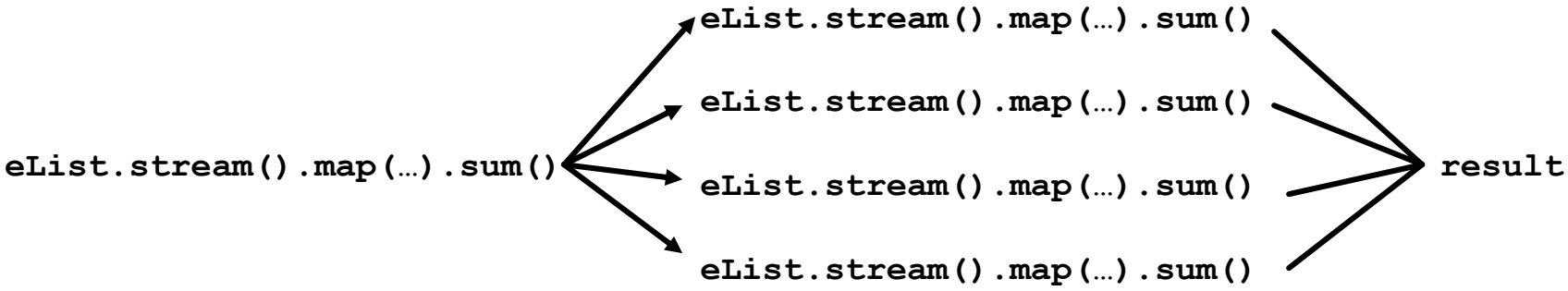
15

Sum

Elements

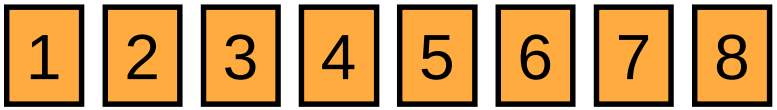
A Look Under the Hood

- Pipeline decomposed into subpipelines.
  - Each subpipeline produces a subresult.
  - Subresults combined into final result.



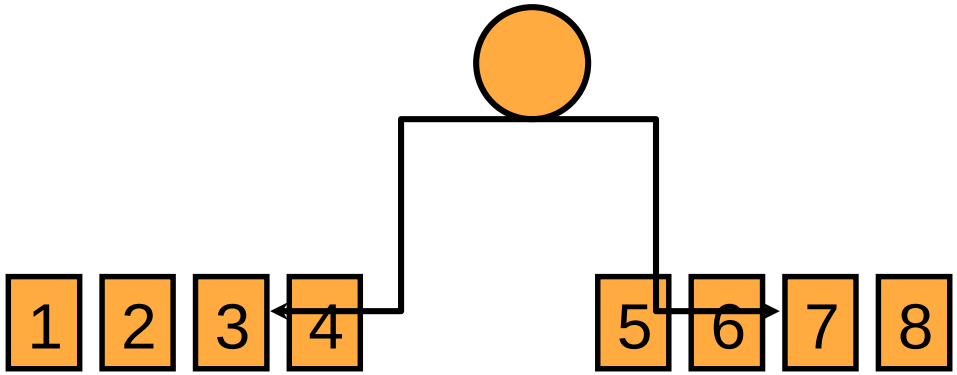
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



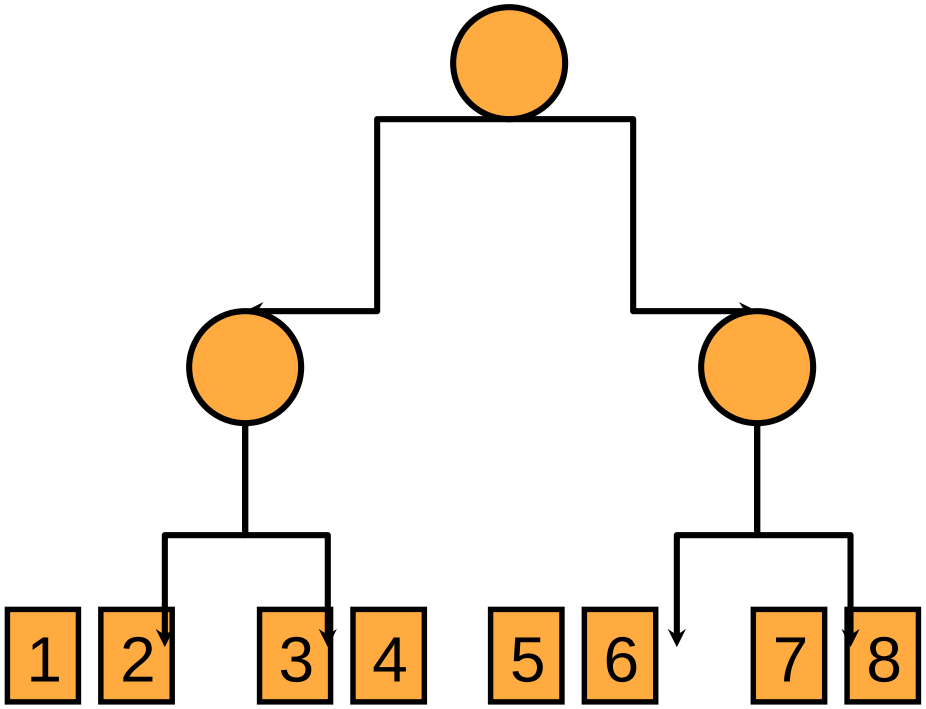
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



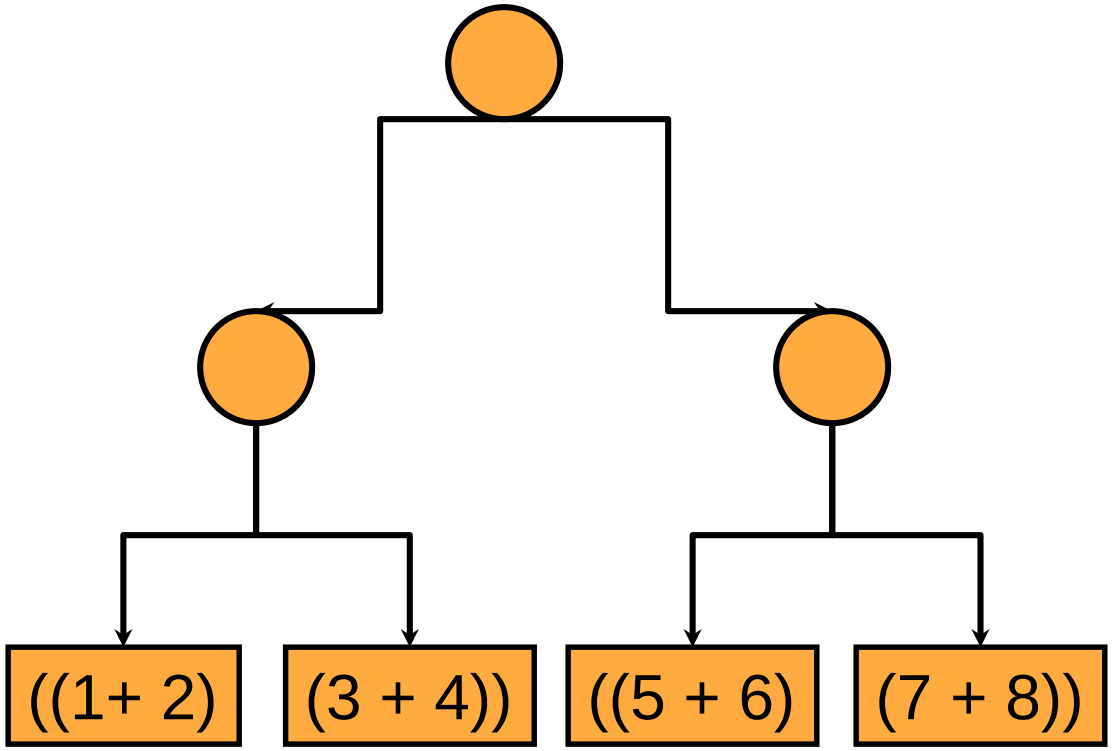
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



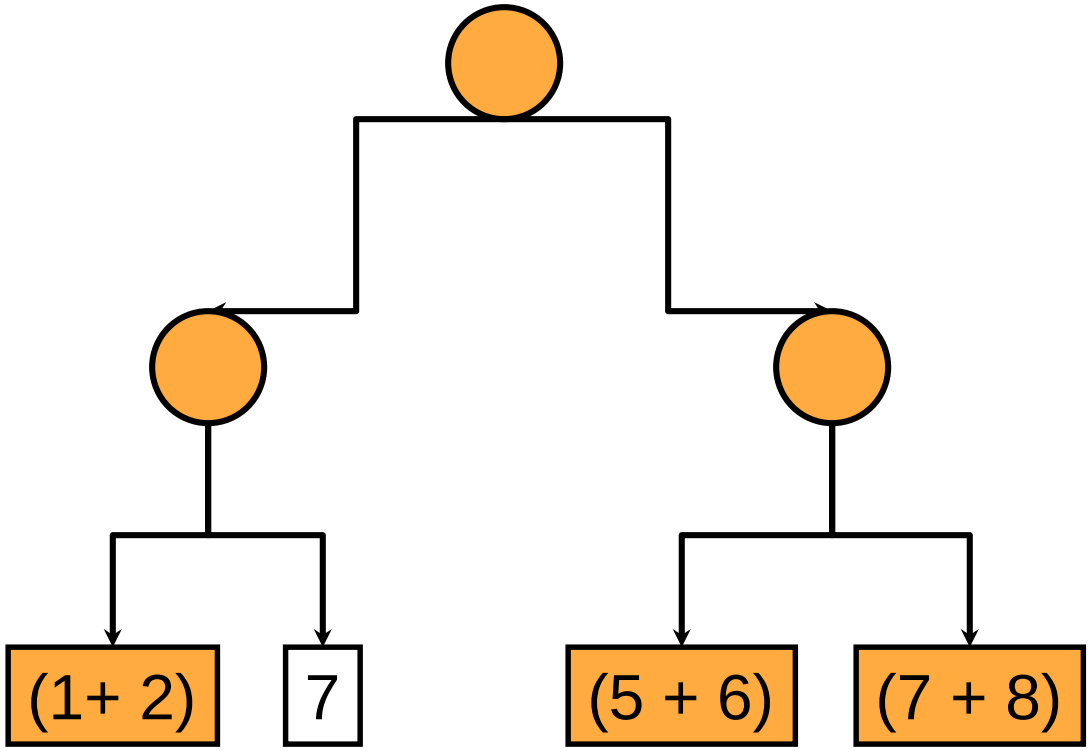
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



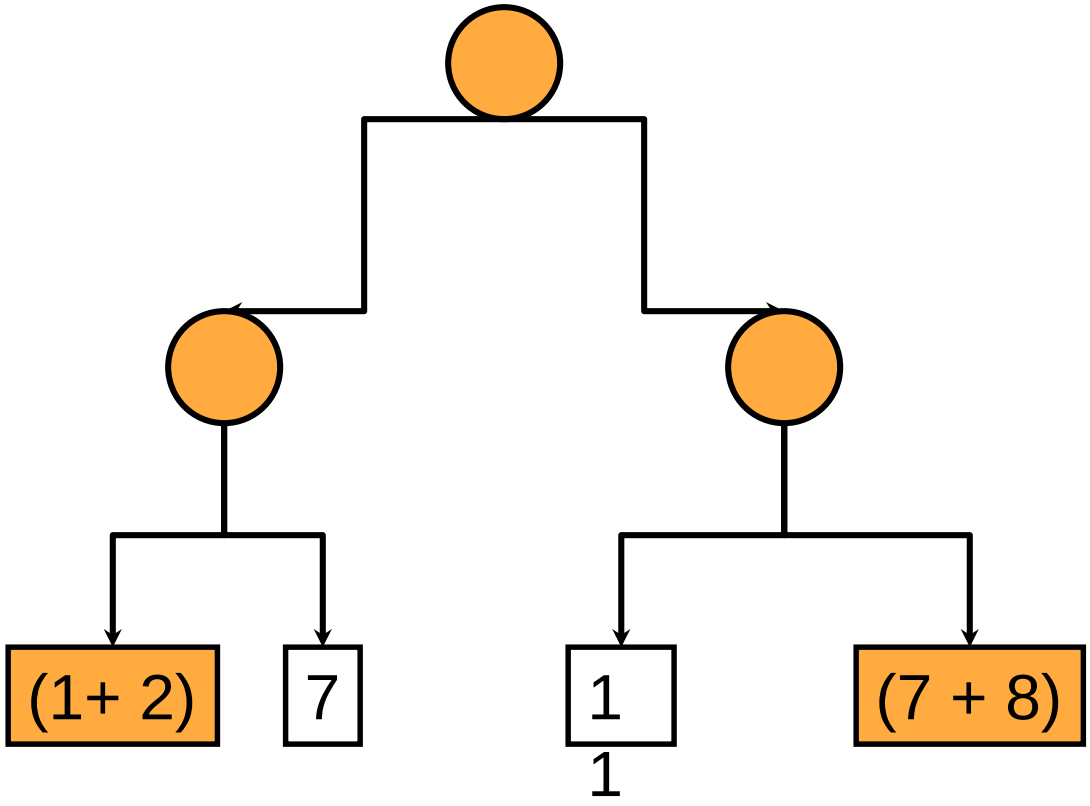
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



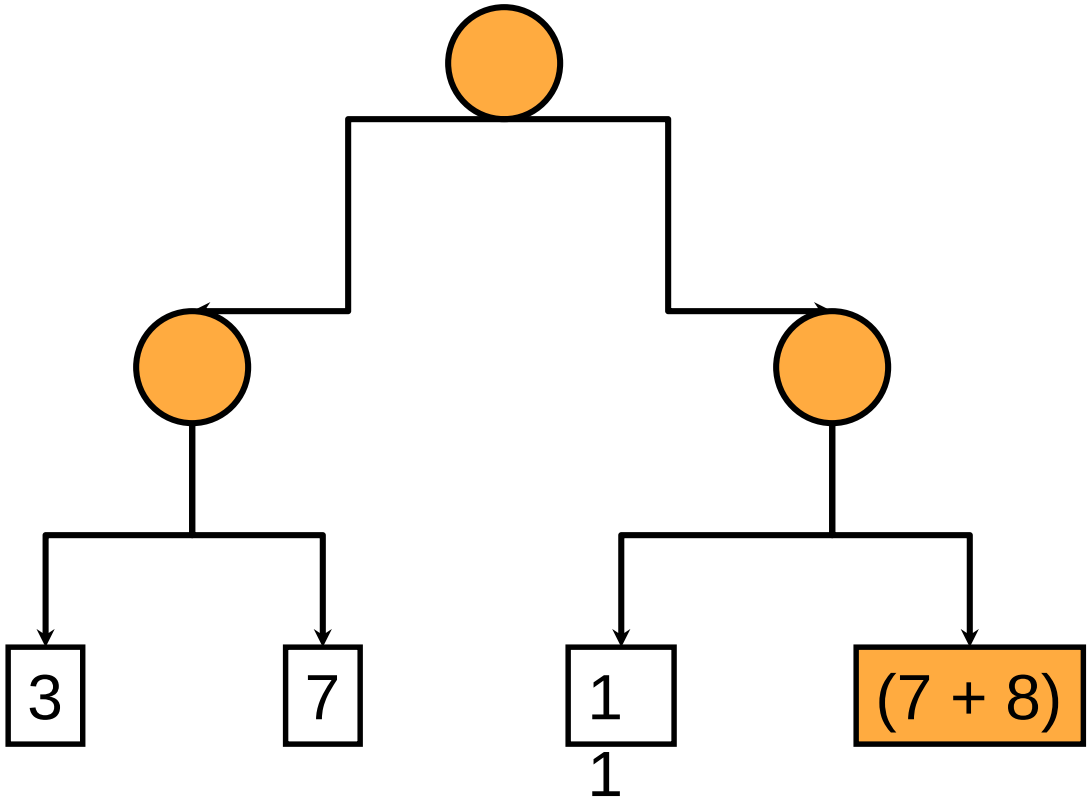
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



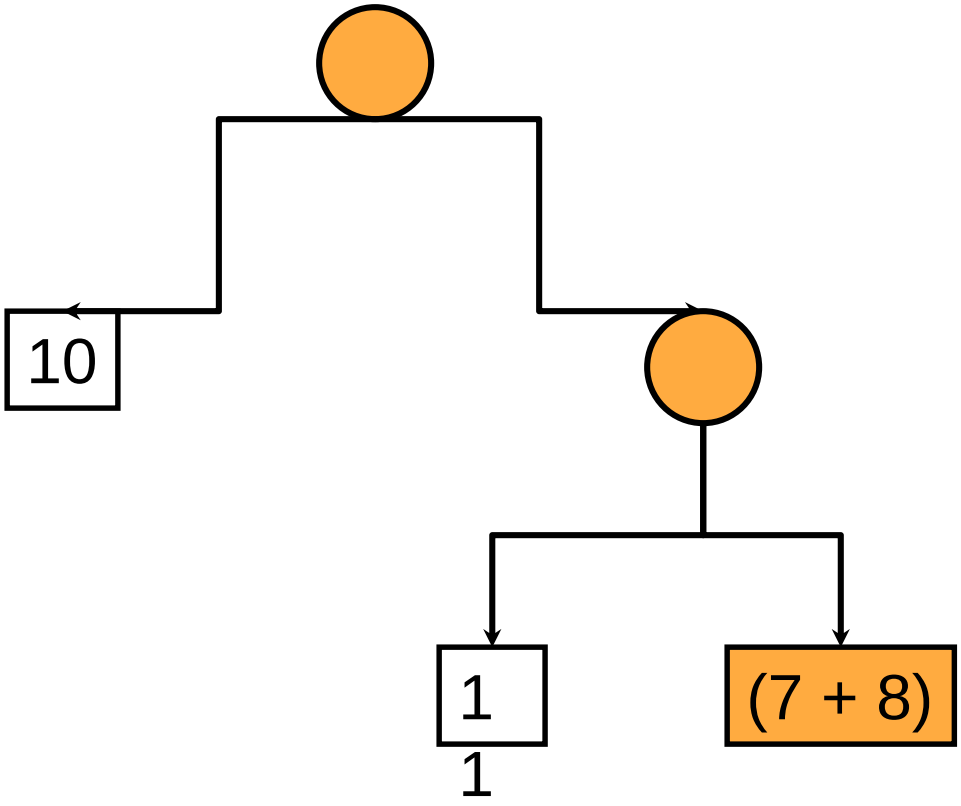
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



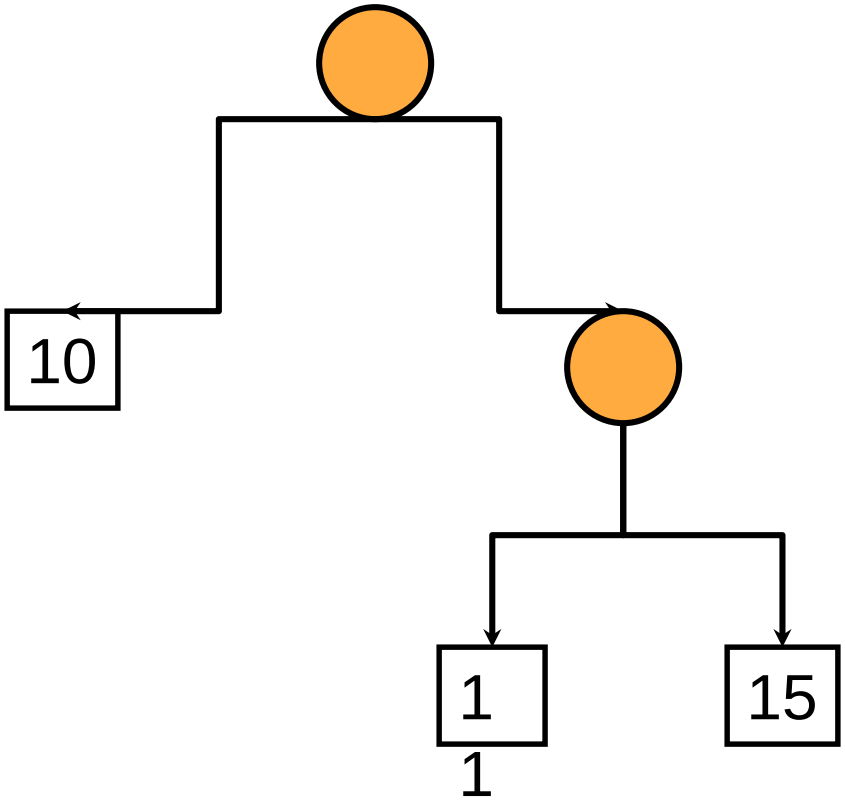
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



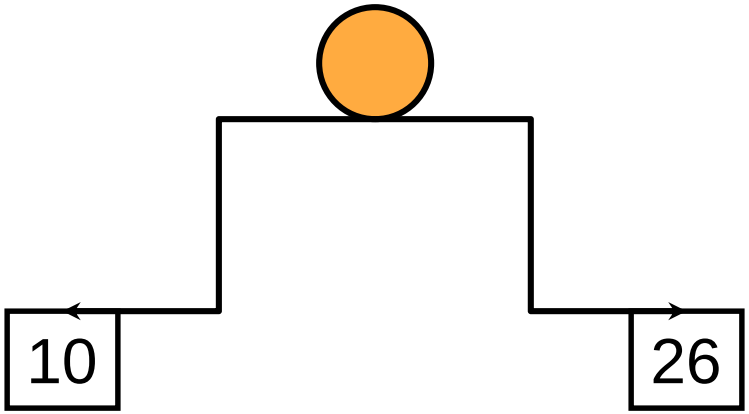
Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```



Illustrating Parallel Execution

```
18          int r2 = IntStream.rangeClosed(1, 8).parallel()
19              .reduce(0, (sum, element) -> sum + element);
```

36

Performance

- Do not assume parallel is always faster.
  - Parallel not always the right solution.
  - Sometimes parallel is slower than sequential.
- Qualitative considerations
  - Does the stream source decompose well?
  - Do terminal operations have a cheap or expensive merge operation?
  - What are stream characteristics?
    - Filters change size for example.
- Primitive streams provided for performance
  - Boxing/Unboxing negatively impacts performance.

A Simple Performance Model

N = Size of the source data set

Q = Cost per element through the pipeline

N * Q ~= Cost of the pipeline

- Larger N*Q -> Higher change of good parallel performance
- Easier to know N than Q
- You can reason qualitatively about Q
  - Simple pipeline example
    - N > 10K. Q=1
    - Reduction using sum
  - Complex pipelines might
    - Contain filters
    - Contain limit operation
    - Complex reduction using `groupingBy()`

Summary

In this lesson, you should have learned how to:

- Review the key characteristics of streams
- Contrast old style loop operations with streams
- Describe how to make a stream pipeline execute in parallel
- List the key assumptions needed to use a parallel pipeline
- Define reduction
- Describe why reduction requires an associative function
- Calculate a value using reduce
- Describe the process for decomposing and then merging work
- List the key performance considerations for parallel streams

Practice

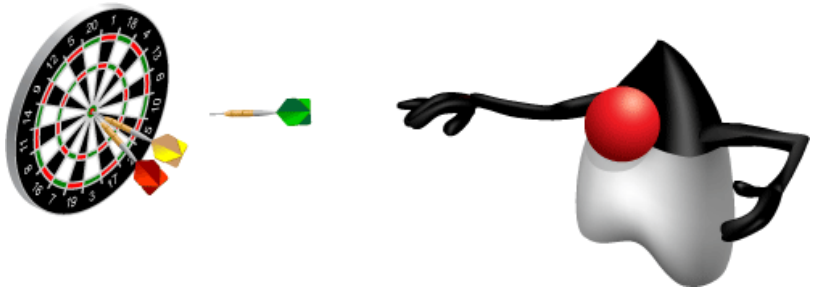
- Lab 17-1: Calculate Total Sales Without a Pipeline
- Lab 17-2: Calculate Sales Totals Using Parallel Streams
- Lab 17-3: Calculate Sales Totals Using Parallel Streams and Reduce

Building Database Applications with JDBC

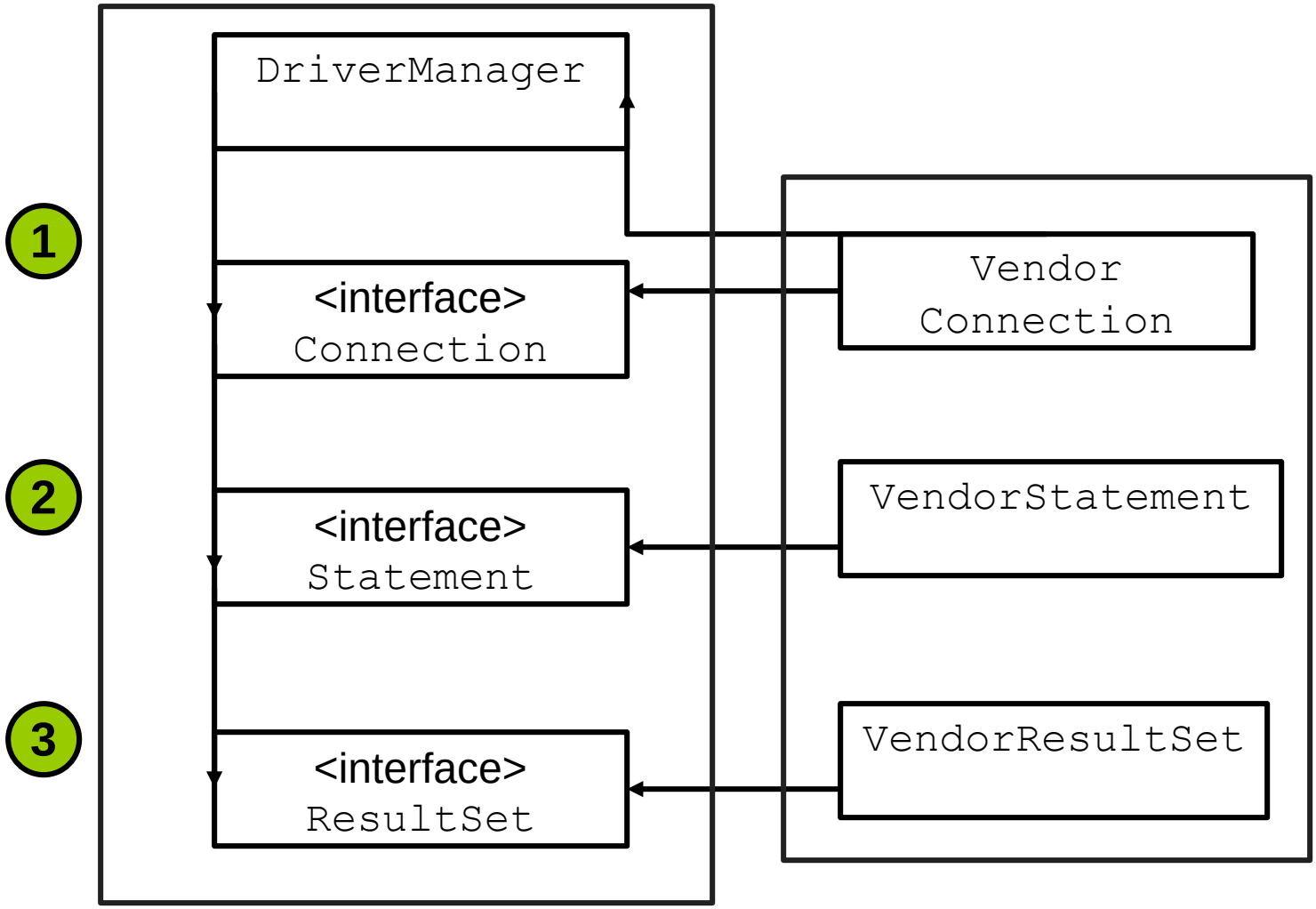
Objectives

After completing this lesson, you should be able to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



Using the JDBC API



`java.sql` class and interfaces

Vendor-Specific JAR File

Using a Vendor's Driver Class

The `DriverManager` class is used to get an instance of a `Connection` object by using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

```
Connection con = DriverManager.getConnection(url,
   username, password);
```

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

```
Statement stmt = con.createStatement();
```

- `java.sql.ResultSet`: An object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the `Statement` object.

```
Statement stmt = con.createStatement();
```

- Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery (query);
```

- Note that there are three Statement execute methods:

Method	Returns	Used for
`executeQuery(sqlString)`	`ResultSet`	`SELECT` statement
`executeUpdate(sqlString)`	`int` (rows affected)	`INSERT`, `UPDATE`, `DELETE`, or a DDL
`execute(sqlString)`	`boolean` (true if there was a `ResultSet`)	Any SQL command or commands

Using a `ResultSet` Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

ResultSet cursor →

The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

rs.next() →

110	Troy	Hammer	1965-03-31	102109.15
123	Michael	Walton	1986-08-25	93400.20
201	Thomas	Fitzpatrick	1961-09-22	75123.45
101	Abhijit	Gopali	1956-06-01	70000.00

rs.next() →

rs.next() →

rs.next() →

`null`

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

CRUD Operations Using JDBC API: Retrieve

```
package com.example.text;

import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

public class SimpleJDBCTest {

   public static void main(String[] args) {
       String url = "jdbc:derby://localhost:1527/EmployeeDB";
       String username = "public";
       String password = "tiger";
       String query = "SELECT * FROM Employee";
       try (Connection con =
            DriverManager.getConnection (url, username, password);
            Statement stmt = con.createStatement ();
            ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password are just for this simple example.

CRUD Operations Using JDBC: Retrieve

Loop through all of the rows in the `ResultSet`.

```
            while (rs.next()) {
                int empID = rs.getInt("ID");
                String first = rs.getString("FirstName");
                String last = rs.getString("LastName");
                Date birthDate = rs.getDate("BirthDate");
                float salary = rs.getFloat("Salary");
                System.out.println("Employee ID:   " + empID + "\n"
                + "Employee Name: " + first + " " + last + "\n"
                + "Birth Date:    " + birthDate + "\n"
                + "Salary:        " + salary);
            } // end of while
        } catch (SQLException e) {
            System.out.println("SQL Exception: " + e);
        } // end of try-with-resources
    }
}
```

CRUD Operations Using JDBC API: Create

```
public class InsertJDBCExample {
    public static void main(String[] args) {
        // Create the "url"
        // assume database server is running on the localhost
        String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "scott";
        String password = "tiger";
try (Connection con = DriverManager.getConnection(url, username,
password))
{
 Statement stmt = con.createStatement();
 String query = "INSERT INTO Employee VALUES (500, 'Jill',
'Murray','1950-09-21', 150000)";
if (stmt.executeUpdate(query) > 0) {
   System.out.println("A new Employee record is added");
   }
String query1="select *  from Employee";
ResultSet rs = stmt.executeUpdate(query1);
//code to display the rows
}
```

Query to insert a row in the Employee.

CRUD Operations Using JDBC API: Update

```
 public class UpdateJDBCExample {
    public static void main(String[] args) {
        // Create the "url"
        // assume database server is running on the localhost
        String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "scott";
        String password = "tiger";
    try (Connection con = DriverManager.getConnection(url, username,
password)) {
            Statement stmt = con.createStatement();
          query = "Update Employee SET salary= 200000 where id=500";
            if (stmt.executeUpdate(query) > 0) {
  System.out.println("An existing employee record was updated
successfully!");
         }
           String query1="select * from Employee";
            ResultSet rs = stmt.executeQuery(query1);
     //code to display the records//
}
```

CRUD Operations Using JDBC API: Delete

```
public class DeleteJDBCExample {

    public static void main(String[] args) {
       String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "scott";
        String password = "tiger”;
   try (Connection con = DriverManager.getConnection(url, username,
password)) {
         Statement stmt = con.createStatement();
         String query = "DELETE FROM Employee where id=500";
   if (stmt.executeUpdate(query) > 0) {
   System.out.println("An employee record was deleted successfully");
   }
       String query1="select * from Employee";
       ResultSet rs = stmt.executeQuery(query1);
```

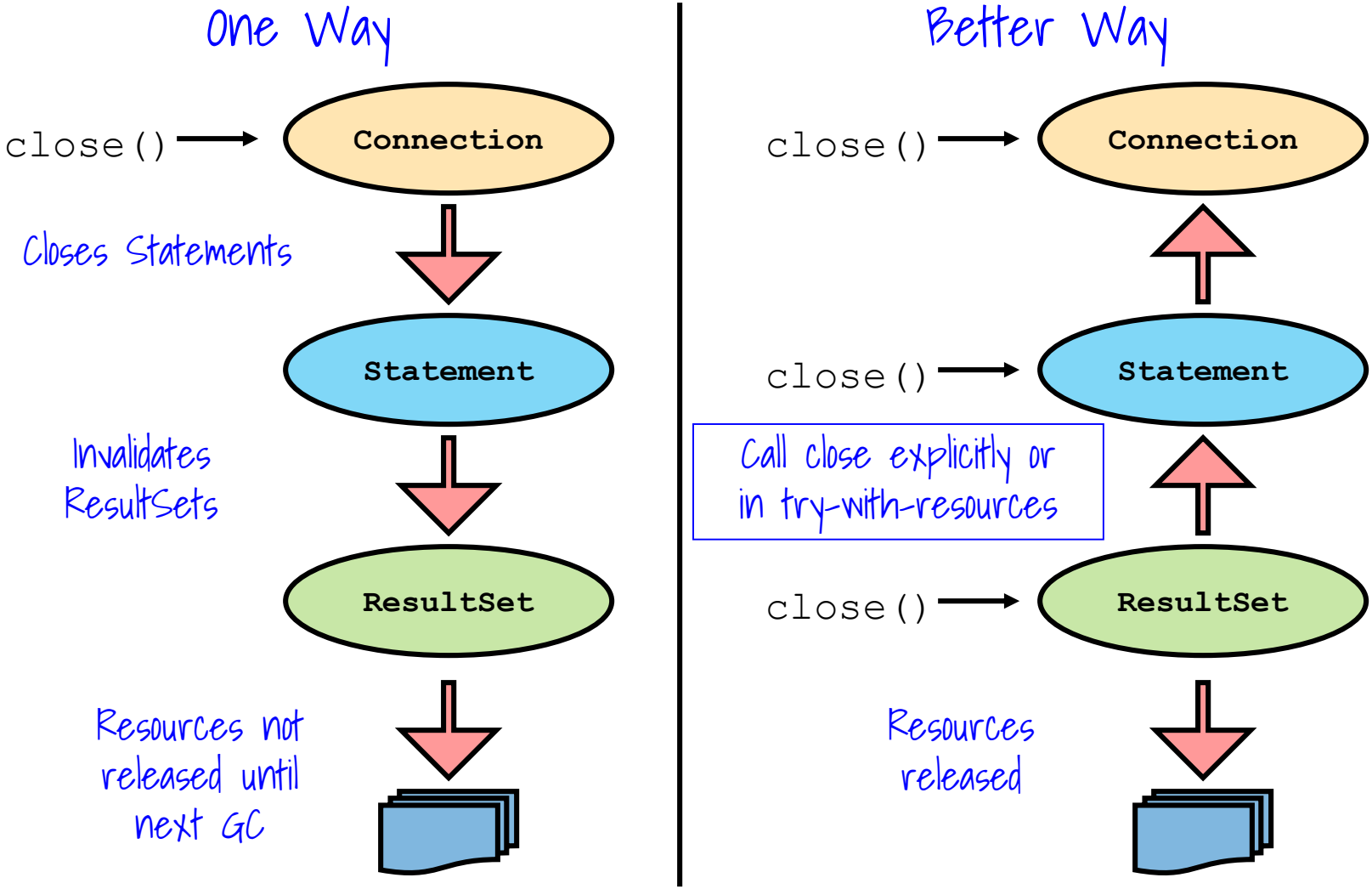
`SQLException` Class

`SQLException` can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the `SQLException`s thrown:

```
catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("Error Code:" + ex.getErrorCode());
        System.out.println("Message:   " + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null) {
            System.out.println("Cause:" + t);
            t = t.getCause();
        }
        ex = ex.getNextException();
    }
}
```

Vendor-dependent state codes, error codes, and messages

Closing JDBC Objects



`try`-with-resources Construct

Given the following `try`-with-resources statement:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The `close()` method is automatically called at the end of the `try` block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the `try` block, separated by semicolons.

Using `PreparedStatement`

`PreparedStatement` is a subclass of `Statement` that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Parameter for substitution.

Substitutes `value` for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- `PreparedStatement` is useful when you want to execute a SQL statement multiple times.

Using `PreparedStatement`: Setting Parameters

In general, there is a **`setXXX`** method for each type in the Java programming language.

`setXXX` arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

Executing `PreparedStatement`

In general, there is a **`setXXX`** method for each type in the Java programming language.

`setXXX` arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

PreparedStatement: Using a Loop to Set Values

```
PreparedStatement updateEmp;
    String updateString = "update Employee"
     + "set SALARY= ? where EMP_NAME like ?";
    updateEmp  = con.prepareStatement(updateString);
    int[] salary = {1750, 1500, 6000, 1550, 9050};
    String[] names = {"David", "Tom", "Nick",
"Harry", "Mark"};
    for(int i:names)
     {
       updateEmp.setInt(1, salary[i]);
       updateEmp.setString(2, names[i]);
       updateEmp.executeUpdate();
    }
```

Using `CallableStatement`

A `CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

The `IN` parameter is passed in to the stored procedure.

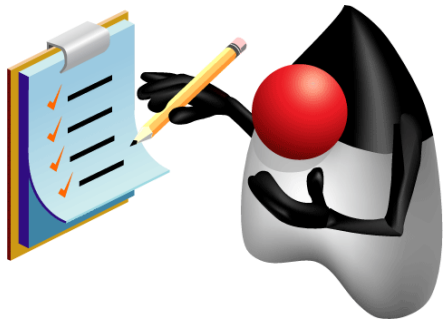
The `OUT` parameter is returned from the stored procedure.

- Stored procedures are executed on the database.

Summary

In this lesson, you should have learned how to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



Lab 18-1 Overview: Working with the Derby Database and JDBC

This Lab covers the following topics:

- Starting the JavaDB (Derby) database from within NetBeans IDE
- Populating the database with data (the Employee table)
- Running SQL queries to look at the data
- Compiling and running the sample JDBC application

Quiz

Which `Statement` method executes a SQL statement and returns the number of rows affected?

a. `stmt.execute(query);`

b. `stmt.executeUpdate(query);`

c. `stmt.executeQuery(query);`

d. `stmt.query(query);`

Quiz

When using a `Statement` to execute a query that returns only one record, it is not necessary to use the `ResultSet`'s `next()` method.

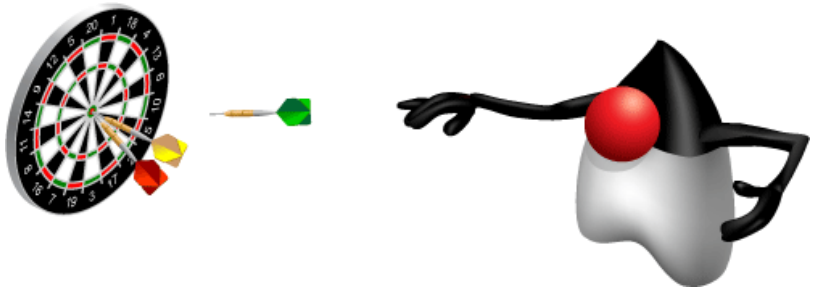
a. True
b. False

Localization

Objectives

After completing this lesson, you should be able to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle



Why Localize?

The decision to create a version of an application for international use often happens at the start of a development project.

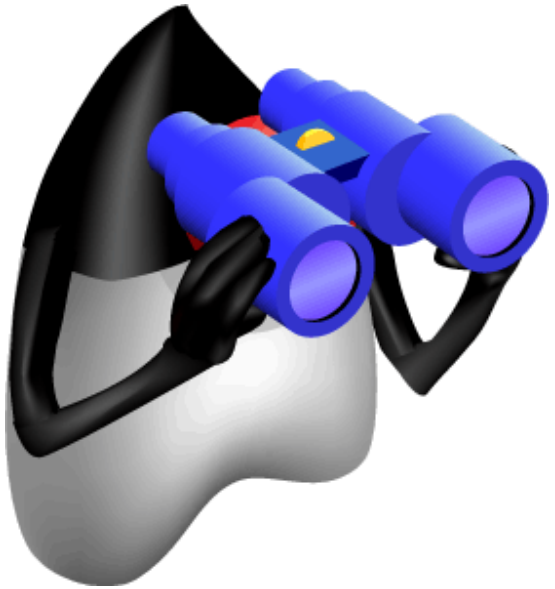
- Region- and language-aware software
- Dates, numbers, and currencies formatted for specific countries
- Ability to plug in country-specific data without changing code

A Sample Application

Localize a sample application:

- Text-based user interface
- Localize menus
- Display currency and date localizations

```
=== Localization App ===
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:
```



`Locale`

A `Locale` specifies a particular language and country:

- Language
  - An alpha-2 or alpha-3 ISO 639 code
  - “en” for English, “es” for Spanish
  - Always uses lowercase
- Country
  - Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code
  - "US" for United States, "ES" for Spain
  - Always uses uppercase
- See the Java Tutorials for details of all standards used.

Properties

- The `java.util.Properties` class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file:

```
hostName = www.example.com
userName = user
password = pass
```

- File name ends in `.properties`.
- File can be anywhere that compiler can find it.

Loading and Using a Properties File

```
public static void main(String[] args) {
  Properties myProps = new Properties();
  try {
    FileInputStream fis = new FileInputStream("ServerInfo.properties");
    myProps.load(fis);
  } catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
  }

   // Print Values
   System.out.println("Server: " + myProps.getProperty("hostName"));
   System.out.println("User: " + myProps.getProperty("userName"));
   System.out.println("Password: " + myProps.getProperty("password"));
}
```

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the –D option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values:

```
java -Dusername=user myApp
```

- Get the `Properties` data from the `System` object:

```
String userName = System.getProperty("username");
```

Resource Bundle

- The `ResourceBundle` class isolates locale-specific data:
  - Returns key/value pairs stored separately
  - Can be a class or a `.properties` file
- Steps to use:
  - Create bundle files for each locale.
  - Call a specific locale from your application.

Resource Bundle File

- Properties file contains a set of key-value pairs.
  - Each key identifies a specific application component.
  - Special file names use language and country codes.
- Default for sample application:
  - Menu converted into resource bundle

MessageBundle.properties

```
menu1 = Set to English
menu2 = Set to French
menu3 = Set to Chinese
menu4 = Set to Russian
menu5 = Show the Date
menu6 = Show me the money!
menuq = Enter q to quit
```

Sample Resource Bundle Files

Samples for French and Chinese

MessagesBundle_fr_FR.properties

```
menu1 = Régler à l'anglais
menu2 = Régler au français
menu3 = Réglez chinoise
menu4 = Définir pour la Russie
menu5 = Afficher la date
menu6 = Montrez-moi l'argent!
menuq = Saisissez q pour quitter
```

MessagesBundle_zh_CN.properties

```
menu1 = 设置为英语
menu2 = 设置为法语
menu3 = 设置为中文
menu4 = 设置到俄罗斯
menu5 = 显示日期
menu6 = 显示我的钱！
menuq = 输入q退出
```

Initializing the Sample Application

```
PrintWriter pw = new PrintWriter(System.out, true);
    // More init code here

    Locale usLocale = Locale.US;
    Locale frLocale = Locale.FRANCE;
    Locale zhLocale = new Locale("zh", "CN");
    Locale ruLocale = new Locale("ru", "RU");
    Locale currentLocale = Locale.getDefault();

    ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle",
     currentLocale);

    // more init code here

    public static void main(String[] args){
        SampleApp ui = new SampleApp();
        ui.run();
    }
```

Sample Application: Main Loop

```
public void run(){
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```

The `printMenu` Method

Instead of text, a resource bundle is used.

- `messages` is a resource bundle.
- A key is used to retrieve each menu item.
- Language is selected based on the `Locale` setting.

```
public void printMenu(){
    pw.println("=== Localization App ===");
    pw.println("1. " + messages.getString("menu1"));
    pw.println("2. " + messages.getString("menu2"));
    pw.println("3. " + messages.getString("menu3"));
    pw.println("4. " + messages.getString("menu4"));
    pw.println("5. " + messages.getString("menu5"));
    pw.println("6. " + messages.getString("menu6"));
    pw.println("q. " + messages.getString("menuq"));
    System.out.print(messages.getString("menucommand")+" ");
 }
```

Changing the `Locale`

To change the `Locale`:

- Set `currentLocale` to the desired language.
- Reload the bundle by using the current locale.

```
public void setFrench(){
    currentLocale = frLocale;
    messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);
}
```

Sample Interface with French

After the French option is selected, the updated user interface looks like the following:

```
=== Localization App ===
1. Régler à l'anglais
2. Régler au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```

Format Date and Currency

- Numbers can be localized and displayed in their local format.
- Special format classes include:
  - `java.time.format.DateTimeFormatter`
  - `java.text.NumberFormat`
- Create objects using `Locale`.

Displaying Currency

- Format currency:
  - Get a currency instance from `NumberFormat`.
  - Pass the `Double` to the `format` method.

- Sample currency output:

```
1 000 000 руб. ru_RU
1 000 000,00 € fr_FR
￥1,000,000.00 zh_CN
£1,000,000.00 en_GB
```

Formatting Currency with NumberFormat

```
 1 package com.example.format;
 2
 3 import java.text.NumberFormat;
 4 import java.util.Locale;
 5
 6 public class NumberTest {
 7
 8   public static void main(String[] args) {
 9
10     Locale loc = Locale.UK;
11     NumberFormat nf = NumberFormat.getCurrencyInstance(loc);
12     double money = 1_000_000.00d;
13
14     System.out.println("Money: " + nf.format(money) + " in
Locale: " + loc);
15   }
16 }
```

Displaying Dates

- Format a date:
  - Get a `DateTimeFormatter` object based on the `Locale`.
  - From the LocalDateTime variable, call the `format` method passing the formatter.
- Sample dates:

```
20 juil. 2011 fr_FR
20.07.2011 ru_RU
```

Displaying Dates with `DateTimeFormatter`

```
 3 import java.time.LocalDateTime;
 4 import java.time.format.DateTimeFormatter;
 5 import java.time.format.FormatStyle;
 6 import java.util.Locale;
 7
 8 public class DateFormatTest {
 9   public static void main(String[] args) {
10
11     LocalDateTime today = LocalDateTime.now();
12     Locale loc = Locale.FRANCE;
13
14     DateTimeFormatter df =
15       DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
16         .withLocale(loc);
17     System.out.println("Date: " + today.format(df)
18         + " Locale: " + loc.toString());
19   }
```

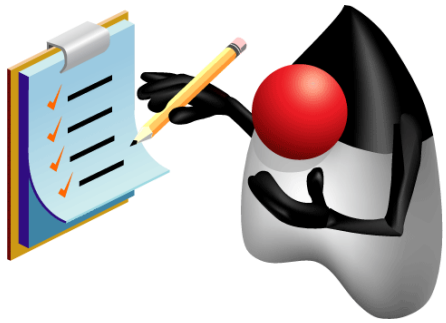
Format Styles

- `DateTimeFormatter` uses the `FormatStyle` enumeration to determine how the data is formatted.
- Enumeration values
  - SHORT: Is completely numeric, such as 12.13.52 or 3:30 pm
  - MEDIUM: Is longer, such as Jan 12, 1952
  - LONG: Is longer, such as January 12, 1952 or 3:30:32 pm
  - FULL: Is completely specified date or time, such as Tuesday, April 12, 1952 AD or 3:30:42 pm PST

Summary

In this lesson, you should have learned how to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle



Lab 19-1 Overview:
Creating a Localized Date Application

This Lab covers creating a localized application that displays dates in a variety of formats.

Quiz

Which bundle file represents a language of Spanish and a country code of US?

a. `MessagesBundle_ES_US.properties`
b. `MessagesBundle_es_es.properties`
c. `MessagesBundle_es_US.properties`
d. `MessagesBundle_ES_us.properties`

Quiz

Which date format constant provides the most detailed information?

a. LONG
b. FULL
c. MAX
d. COMPLETE