
CSYE 6200

Concepts of Object Oriented Design

Introduction References

Daniel Peters

d.peters@neu.edu

Resources on the Web

- IntelliJ (useful for installing JDK)
- Eclipse IDE for Java Developers
<https://www.eclipse.org/downloads/packages/>
- Apache NetBeans
<https://netbeans.apache.org/>

Old Resources on the Web

- Java Development Kit (JDK)

<https://www.oracle.com/java/technologies/jdk8-downloads.html>

- NetBeans IDE

<https://netbeans.org/downloads/8.2/>

Resources on the Web

- jGRASP

<https://www.jgrasp.org/>

Resources on the Web

- Online IDE

[https://www.tutorialspoint.com/
compile_java_online.php](https://www.tutorialspoint.com/compile_java_online.php)

- Online Compiler

[https://www.onlinegdb.com/
online_java_compiler](https://www.onlinegdb.com/online_java_compiler)

-

Resources (Cont'd)

- Java Standard Edition 8 Documentation

<http://docs.oracle.com/javase/8/docs/api/>

- Java SE 8 Documentation Download

<http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html>

- Java Tutorial

<http://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html>

Resources (Cont'd)

- Oracle: Java A Beginners Guide (chapter 1)
<http://www.oracle.com/events/global/en/java-outreach/resources/java-a-beginners-guide-1720064.pdf>
- Eclipse Documentation
<https://eclipse.org/users/>

References

- Reference Books
 - *Java: A Beginner's Guide* by Schildt, 6th Edition, Oracle Press, ISBN-13: 978-0-07-180925-2, ISBN-10: 0-07-180925-2
 - *Java: How to Program* by Deitel, 7-10th Edition, Prentice Hall, ISBN-13: 978-0132575669, ISBN-10: 0132575663
 - *The Complete Reference* by Schildt, 8th Edition, Oracle Press, ISBN-13: 978-0-07-160631-8, MHID: 0-07-160631-9

References (cont'd)

- Reference Books
 - *Thinking in Java* by Bruce Eckel, 4th Edition, Prentice Hall, ISBN-13: 978-0131872486, ISBN-10: 0131872486
 - *Effective Java* by Joshua Bloch, 2nd Edition, Addison-Wesley, ISBN-13: 860-1300201986, ISBN-10: 0321356683
 - *Head First Design Patterns* by Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson, 1st Edition, O'REILLY, ISBN-13: 000-0-596-00712-4, ISBN-10: 0-596-00712-4
-

References (cont'd)

- Reference Books
 - *Head First Java* by Kathy Sierra, Bert Bates, 2nd Edition, O'REILLY, ISBN-13: 978-0-596-00920-5, ISBN-10: 0-596-00920-8

CSYE 6200

Concepts of Object-Oriented Design

Object Oriented (OO) Concepts

Daniel Peters

d.peters@neu.edu

Fundamental Object Oriented

- Class
- Object
- Borrows from real life
 - Life has different things
 - Rock, Tree, Person, Car, School, Education, etc.
 - We use things in life
 - Throw Rock, Drive car, Obtain Education, etc.

Demonstrate OOP in Java

1. Create a file

Driver.java

2. Create Driver class in file Driver.java

```
public class Driver {  
} // useless do nothing class
```

3. Compile class Driver.java

javac Driver.java

4. Execute class Driver (???)

java Driver

Javac Java Compiler

- Java Compiler: **javac**
 - Java is a compiled language
 - Can't execute source code
 - Source file **Driver.java** contains source code
 - **Driver.java** is a text file containing java code
 - Compilation creates class file from source file
 - **Driver.class** is a binary file containing java bytecode

Java: Java Execute Command

- Java command: **java**
 - Used to set up Java run-time environment to execute java class files
 - Java Class files contain *Bytecodes*
 - Bytecodes are interpreted by the *Just In Time* (JIT) compiler in the *Java Virtual Machine (JVM)*

Complete Driver.java

- Create a *complete* Driver class with a class member method main (in file Driver.java)
 - *Note: Driver class is using System class*

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

- Compile class Driver.java: **javac** Driver
 - Execute class Driver: **java** Driver
-

CONSOLE OUTPUT

Hello World!

Class System

- Part of Java Class Library
- Used to interact with user through system console device
 - Output text data to console (standard output) device (**stdout**)
 - Output error text data to console (standard error) device (**stderr**)
 - Input text data from console (standard input) device (**stdin**)

4 Object Oriented Principles

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Class

- Specification used to instantiate object
 - EXCEPTIONS
 - Interface
 - Abstract (pre Java 8)
- Both data and methods ENCAPSULATED together in same class
- Everything required to instantiate one or more object

Class

- Data: state
 - Class (global) data
 - **static**
 - Single copy associated only with class
 - Program Scope:
 - Does not require instantiation: available entire program
 - Instance object data
 - Non-static
 - Unique copy associated with each object
 - Object instance Scope:
 - Requires object instantiation

Class

- Methods operate on member data
 - **static**
 - Associated with the class
 - Does not require object instantiation
 - Non-static
 - Associated with each object
 - Object must be instantiated
 - Abstract
 - Function signature only (pre Java 8)
 - Must be implemented for instantiation
-

Object Oriented Design (OOD)

- Object Oriented Design
 - Objects Model Real Life
 - Think Block Diagram

Object Oriented Design (OOD)

- Object Oriented Design
 - Abstraction
 - Black Box
 - Encapsulation
 - Data and Method Co-located together in same class
 - Inheritance
 - Polymorphism
 - Many forms
 - Objects Model Real Life
 - Think Block Diagram
-

Object Oriented Design (OOD)

- Abstraction (Black Box)
 - Data Hiding
 - Access Modifiers
 - public
 - private
 - Protected (package private)
 - (default)
 - Functionality Hiding
 - Java Concrete class
 - Java Interface
 - Java Abstract class
 - Used for Application Programming Interface (API)

Object Oriented Design (OOD)

- Encapsulation
 - Data and Method Co-located together in same class
 - Private data members
 - » NOT accessible by any other classes
 - Public API methods
 - » API defines how class is usable by other classes

Object Oriented Design (OOD)

- Inheritance
 - Super (Parent) class
 - API
 - Java Interface or abstract class
 - » Specify “What” without “How”
 - A *general* implementation
 - Default implementations
 - » Java concrete or abstract class
 - Derived (child or sub) class
 - A more specific implementation of parent class

Object Oriented Design (OOD)

- Polymorphism (many forms)
 - Inheritance
 - Super class (Parent base class)
 - Sub-class (Derived child class)
 - At run-time, super class variable is used as API and subclass method is called.

Object Oriented Design (OOD)

- Polymorphism (many forms)
 1. Overloaded methods
 - Same name
 - Different signatures
 - Number of parameters
 - Types of parameters
 2. Overridden methods (@Override)
 - Run-time polymorphism
 - Derived child class overrides method in parent base class (super class) to provide customized method

OOD Benefits

- Benefits of Object-Oriented Design
 - Simplicity: readable code
 - Flexibility: easy to refactor and change
 - Scalability: adaptable
 - Testability: unit test
 - Maintainability: easy to fix bugs
 - Design Decomposition
 - From complex problem to basic components
 - Phased development effort
 - Distributed Team development

UML Basics

Object Management Group
Unified Modeling Language

Introduction

Daniel Peters

d.peters@neu.edu

UML

- Unified Modeling Language
 - UML 1.0 draft created by Object Management Group in January 1997
<http://www.uml.org/>

UML

- Unified Modeling Language
 - A picture language
 - Visualize
 - Specify/Document
 - Construct

UML

- Unified Modeling Language

Object Oriented Analysis

- Identify Objects in System
- Object inter relationships
- Object Design/Usage

Diagrams

- Diagrams
 - Class diagram
 - Object diagram
 - Use Case diagram
 - Sequence diagram

UML

- Unified Modeling Language

Object Oriented Analysis

- Identify Objects in System
- Object inter relationships
- Object Design/Usage

Relationships

- Association
 - Aggregation
 - “Has-A”
 - Composition
 - “Has-A”
- Generalization
 - Inheritance
 - “Is-A”

Association Relationship

- Association: Has-A Relationship
 - Aggregation
 - Weak association: linked but independent objects
 - possessed object can survive after possessing object
 - Line with open arrowhead outline
 - Composition
 - Strong association: linked objects are dependent
 - possessed object is destroyed along with possessing object
 - Line with solid filled-in arrowhead

Aggregation Association Relationship

- Aggregation Association: *weak*

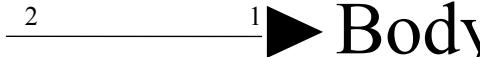
Teacher $\xrightarrow{1..*} 1 \triangleright$ Classroom

Student $\xrightarrow{10..*} 1 \triangleright$ School

Employee $\xrightarrow{1..*} 1 \triangleright$ Corporation

Composition Association Relationship

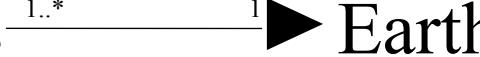
- Composition Association: *strong*

Eyes  Body

```
graph LR; Eyes[Eyes] -- "2" --> Body[Body];
```

Astronauts  Spacecraft

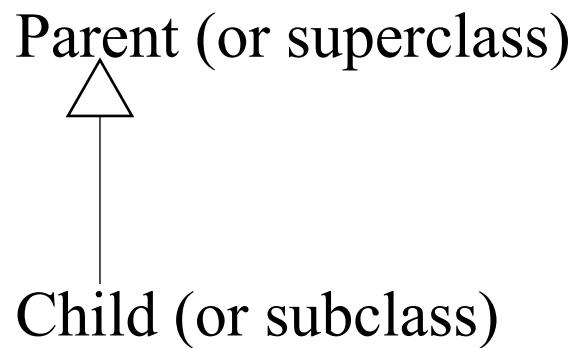
```
graph LR; Astronauts[Astronauts] -- "6..*" --> Spacecraft[Spacecraft];
```

Inhabitants  Earth

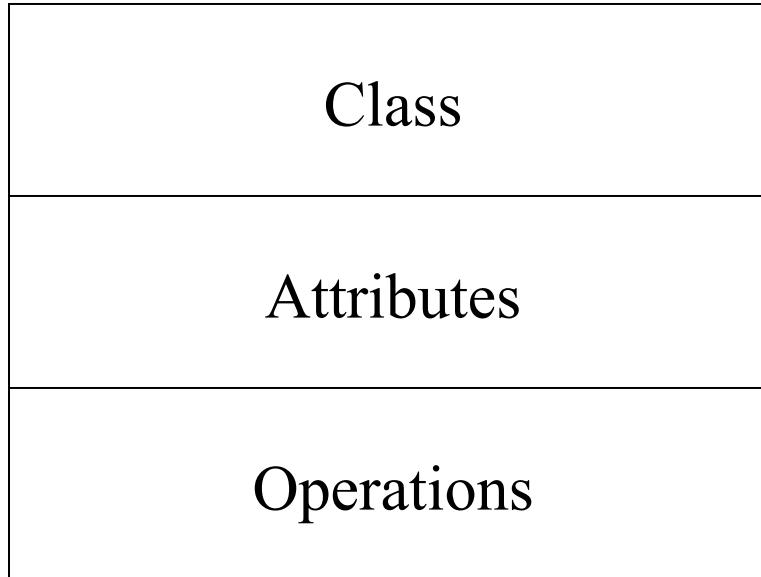
```
graph LR; Inhabitants[Inhabitants] -- "1..*" --> Earth[Earth];
```

Relationships

- Generalization

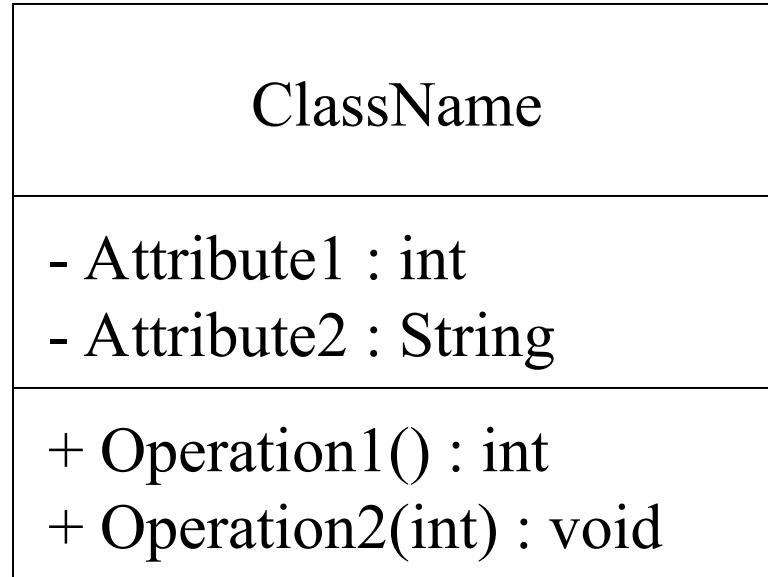


Class Diagram

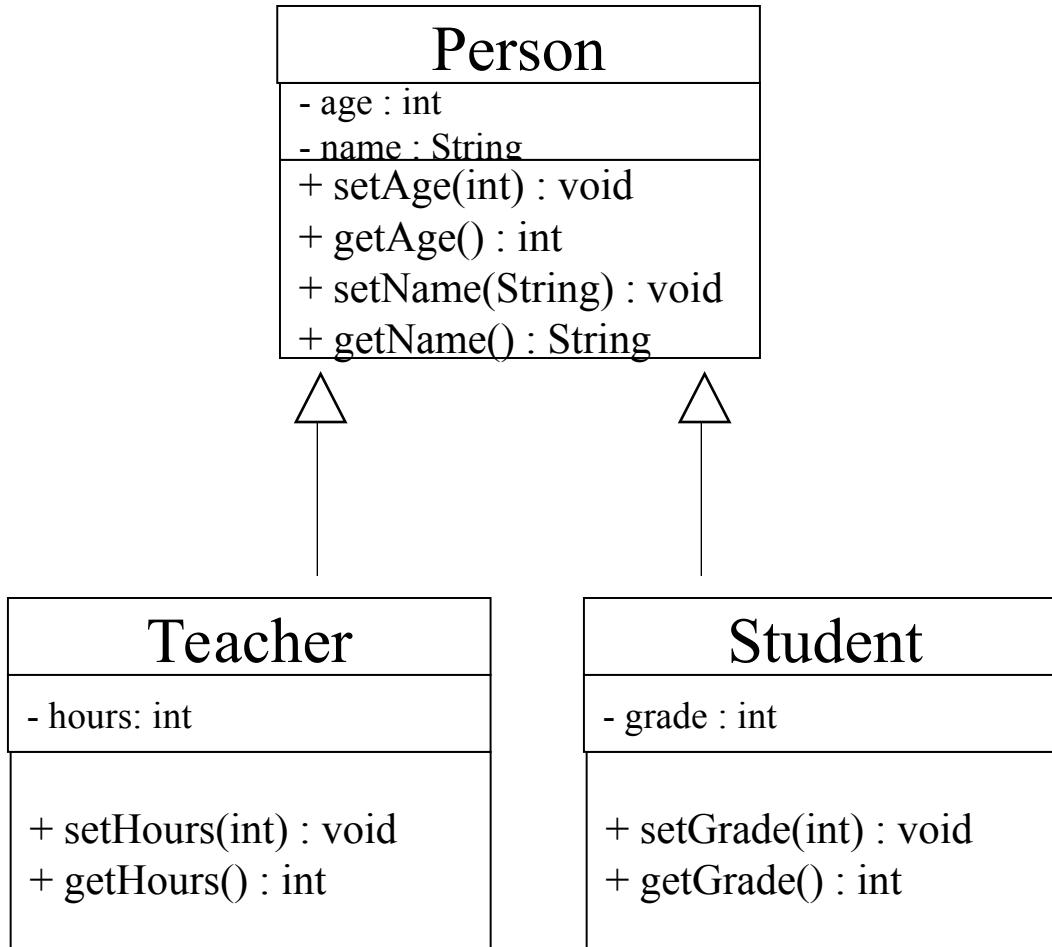


Class Diagram

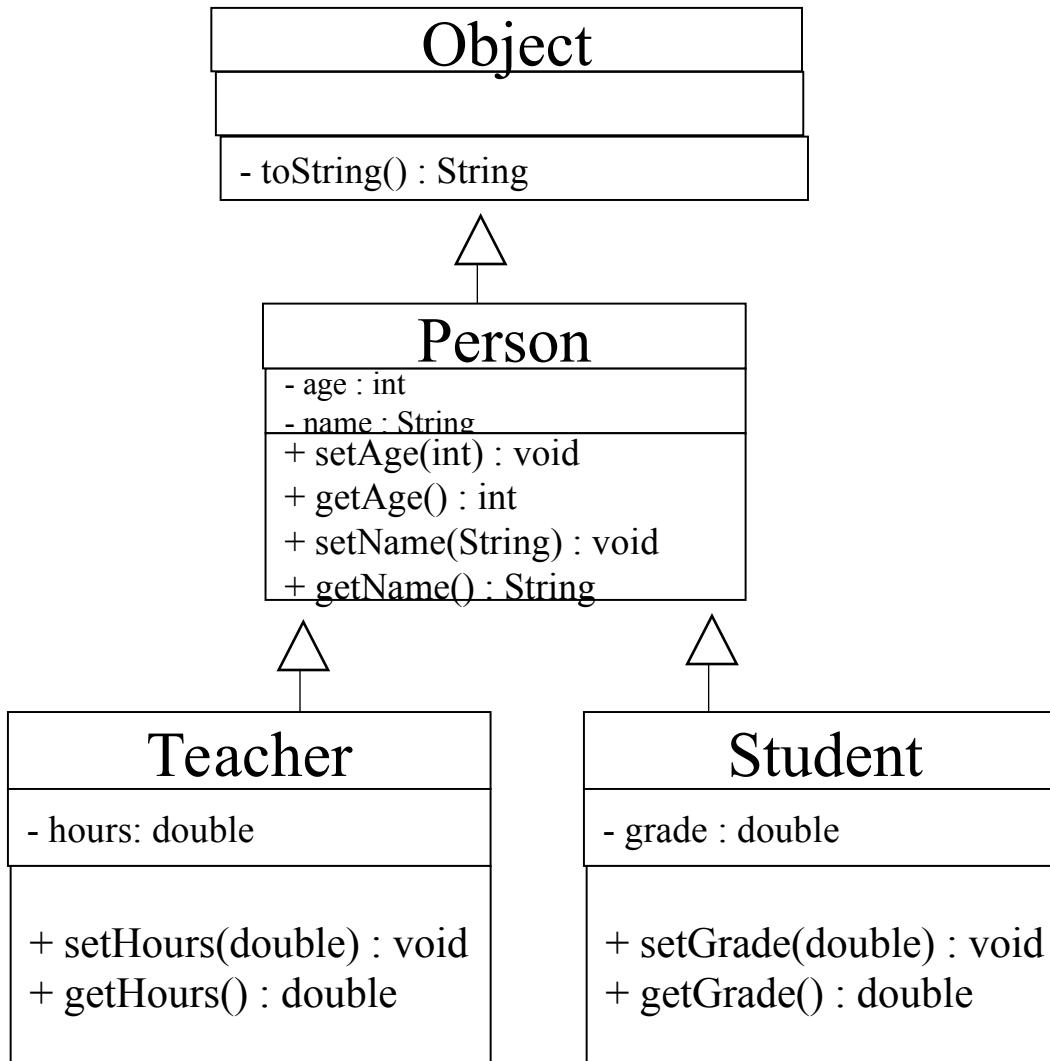
- private
+ public



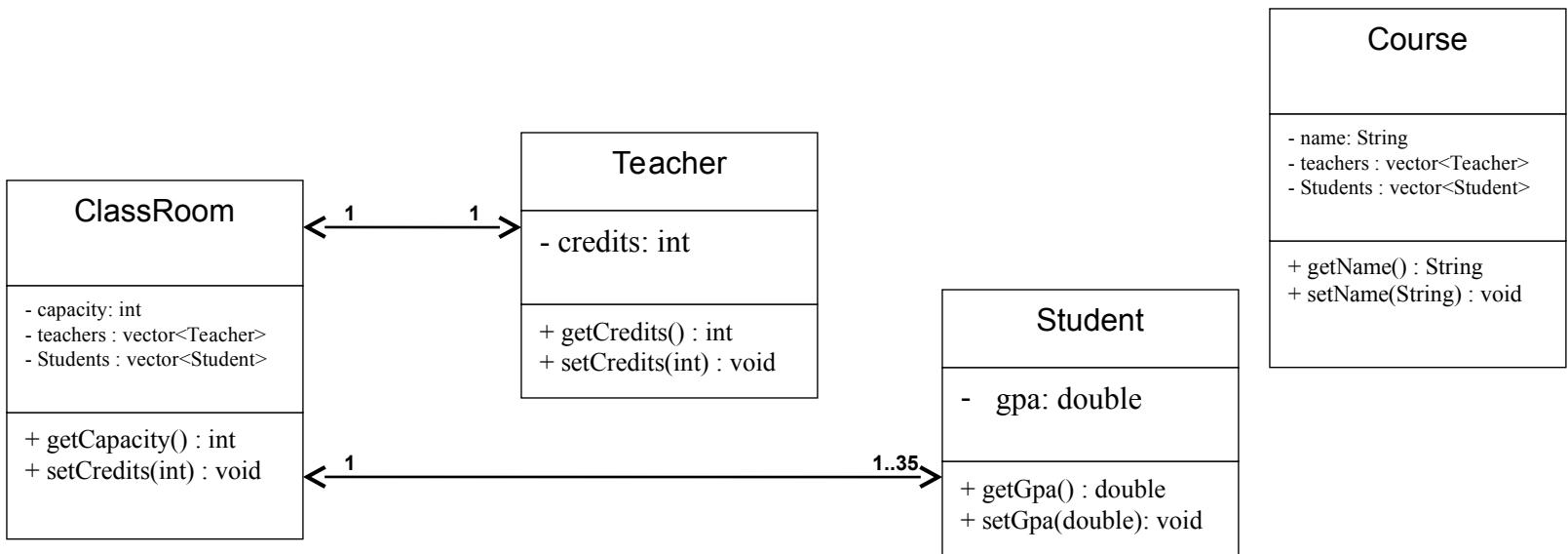
Person Class Diagram



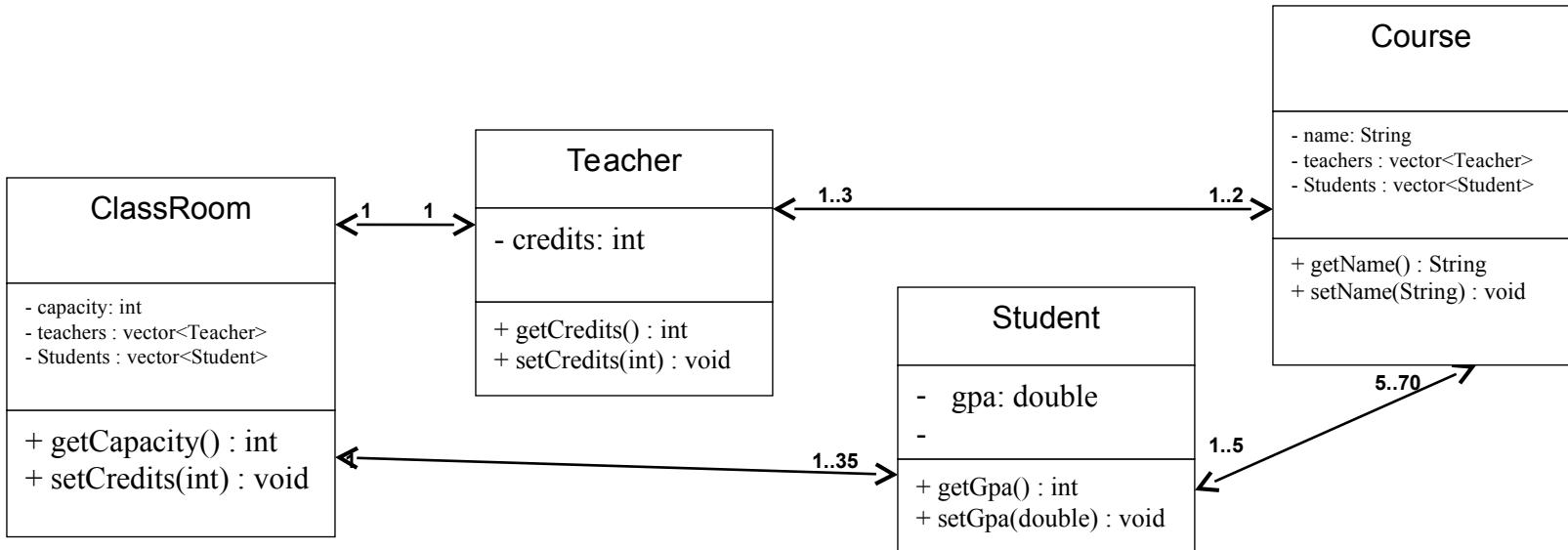
Person Class Diagram



ClassRoom



ClassRoom



CSYE 6200

Concepts of Object Oriented Design

Java Program

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Constants
 - 2. Variables: NOT INITIALIZED BY DEFAULT
 - 1. Char
 - 2. integer
 - 3. float
 - 3. Scope

-
- Java is all about Classes
 - A Class is used to instantiate (i.e., create) one or more Objects

Java program is made up of various classes

- User created classes
- Libraries: Pre-existing classes

- Java Class Libraries
 - Classes organized in packages
 - We ALWAYS create OUR classes in packages
-

-
- Java Program
 - Create Java source (.java) file
 - editor or Integrated Design Environment (IDE)
 - javac Hello.java
 - Compile source file
 - Compiler (javac) creates .class file
 - Load classes (.class) into memory
 - Class loader loads class files from disk

• Java Program

– Verify

- Byte code verifier: checks byte codes in each class for security

– JVM

- Just-in-time (JIT) compiler translates bytecodes into machine specific language for execution

NOTE: Java Program execution begins in the main method

compiled language: ex: java, c++

-> class

-> bytecodes (portable layer - this can be interpreted: in windows, mac, etc)

-> loaded onto jvm

-> executes

prototype languages - interpreted language: ex: python, javascript

-
- Java Types
 - Primitive Types
 - boolean, char, short, int, float, long, double.
 - Stack memory allocation
 - Non-Primitive (Reference) Types
 - Used to reference and hold Objects, e.g. String
 - Heap memory allocation
 - Java is
 - a STRONGLY TYPED LANGUAGE
 - a STATICALLY TYPED LANGUAGE
-

-
- Java is a **STRONGLY TYPED LANGUAGE**
 - All variables are required to have types
 - Java is a **STATICALLY TYPED LANGUAGE**
 - Once assigned a type variables retain that type assignment for duration of program execution
 - EXCEPTION: type casting

-
- Classes are Reference types
 - Class variable holds reference (i.e. pointer) to actual object allocation on heap
 - Class methods (like functions)
 - Each supplied argument must have a type
 - The return value must have types
 - Methods which do not return a value use void

-
- 8 Primitive Types
 - boolean: ONLY true or false
 - byte: 8-bits
 - char: 16-bits
 - short: 16-bits
 - int: 32-bits
 - float: 32-bits
 - long: 64-bits
 - double: 64-bits
-

-
- Online Information
 - Java 8
 - <http://docs.oracle.com/javase/8/docs/api/index.html>
 - Java 7
 - <http://docs.oracle.com/javase/7/docs/api/index.html>

CSYE 6200

Concepts of Object Oriented Design

Java Data Types

Daniel Peters

d.peters@neu.edu

-
- Lecture
 1. Java Language Basics
 2. Java Data Types
 3. Java Primitive Types
 4. Java String class
 5. Java Reference Type
 6. Java Parameter Passing
-

Java Language

- Object Oriented Programming Language
 - Data
 - Memory used by the program
 - Program statements
 - Code instructing the actions of the processor
 - Class
 - Data
 - Methods (program code) operating on class data

Java Language

- Everything is a class
 - Definable aggregate containing data and methods
 - All data and code in Java exists only in context of a class
- Java Language Usage
 - Use class statically
 - Use object instantiated (created) from a class

Java Language

- Statically typed programming language
 - “*The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.*”
- All data must be **declared** and made known to compiler before its first use
- DataType name;**

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Java Language

- Statically typed Languages include:
 - Java
 - C
 - C++
- Dynamically typed languages include:
 - Python
 - Ruby

Java Language

- Strongly typed programming language
 - All data (variables and constants) must **ALWAYS** be **declared** along with its **type**.
 - Identify the memory location by symbol name
 - Identify the memory contents by data type
- Declaration Examples:

Data Type Symbol Name

 - 1.int age;
 - 2.String name;
 - 3.class Person { }

All data members must have
a predefined type

Java Data Type Categories

- Only Two Data Type Categories
 - 1. Pre-Defined Primitive data types:
 - 2. Definable Reference data types:

Java Primitive Data Type

- Primitive data types:
 - Fundamental **predefined** data types
 - Passed by Value
 - Data value is **copied** and passed as a parameter therefore the **original data value cannot be changed when passed by value**

Java Reference Data Type

- Reference data types:
 - Classes and Objects are **definable aggregates**
 - Passed by Reference (like a pointer)
 - Reference is copied and passed as a parameter but always references the **same data object**

” The reference values (often just *references*) are pointers... ”

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.3.1>

Java Primitive Data Types

1. **byte** *8-bit integer* (2^7 to 2^7 minus 1, i.e. -128 to 127)
2. **short** *16 bit integer* (2^{15} to 2^{15} minus 1, i.e. -32,768 to 32,767)
3. **int** *32 bit integer* (- 2^{31} to 2^{31} minus 1)
4. **long** *64 bit integer* (- 2^{63} to 2^{63} minus 1)
5. **float** *32-bit single precision floating point*
6. **double** *64-bit double precision floating point*
7. **boolean** *true or false*
8. **char** *16 bit Unicode character*

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Java Primitive Data Type Use

```
int n = 0; // declare, create, init int value 0
```

```
n = 7; // overwrite int value with 7;
```

```
n++; // increment int value by 1
```

```
n = n + 1; // increment int value by 1
```

Java Primitive Data Types

- “... **new** keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class.“

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Java Primitive Data Types

- Literal values for primitive data types:
 1. byte b = 0;
 2. short s = 1000;
 3. int = 100000;
 4. long x = 0L;
 5. float y = 0.0f
 6. double z = 0.0d
 7. '\u0000' for char
 8. false for boolean

Java Primitive Data Types

- Literal values for primitive data types:

```
int n1 = 13;          // 13 in decimal notation
```

```
int n2 = 0b1101;    // 13 in binary notation
```

```
int n3 = 0x0d;      // 13 in hexadecimal notation
```

```
double x1 = 123.4
```

```
double x2 = 1.234e2    // x1 in scientific  
                      notation
```

Java Primitive Data Types

- Literal values for primitive data types:

```
long creditCardNum = 1234_5678_9012_3456L;  
long socialSecurityNumber = 999_99_9999L;  
float pi = 3.14_15F;
```

Java Primitive Data Types

- Literal values for primitive data types:

```
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;
```

Java Primitive Data Types

- PLACE “ ” ONLY BETWEEN DIGITS

```
int x4 = 0_x52; // INVALID
```

- NEVER At the beginning or end of a number

```
int x2 = 52_ ; // INVALID
```

```
int x5 = 0x_52; // INVALID
```

- NEVER Adjacent to a decimal point in a floating point literal

```
float pi1 = 3_.1415F; // INVALID
```

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

- NEVER Prior to an F or L suffix, example:

```
long socialSecurityNumber1 = 999_99_9999_L;
```

- NEVER In positions where a string of digits is expected

Java Primitive Data Types

- Literal values for primitive data types:

```
char a1 = 'A';           // uppercase A character
char a2 = 'a';           // lowercase a character
char c1 = '\n';          // newline character
char c2 = '\t';          // tab character
```

Java Primitive Data Types

- Default values for primitive data types in class:
 1. 0 for byte
 2. 0 for short
 3. 0 for int
 4. 0L for long
 5. 0.0f for float
 6. 0.0d for double
 7. '\u0000' for char
 8. false for boolean
-

Java Primitive Data Types

- Declaring variables of primitive data types without explicit initialization
 - Compiler set variables to reasonable default value

```
int age;           // initialized to 0
double gpa;       // initialized to 0.0d
char middleInitial; // initialized to '\u0000'
```

Java Primitive Data Types

- Declaring and initializing variables of primitive data types

```
int age = 17;
```

```
double gpa = 4.0;
```

```
char middleInitial = 'G';
```

Java Reference Type

- A Class is a reference type
 - Definable custom data type
 - The fundamental Unit for Java Object Oriented Programming: Everything is a class
 - Wrapper for definable data and/or code
 - Aggregate data type
 - Including Primitive data types
 - Including Other reference types
 - Including Program code

Aggregate data type is used to keep all related data together in a way that emphasizes the relationship of the data. These may include primitive data types, other reference types & other parts of program code.

Class Static members

- Use class statically
 - Class members defined as ‘static’
 - ONE memory allocation
 - on the heap, static members are allocated before runtime at start of the program
 - Program Scope
 - runtime - when object is instantiated
 - Always available for use
 - No need to create with “new”
 - i.e. object need not be instantiated for using static members / class
- All outer classes, constructors are never static
- Inner classes may or may not be static or public or private

Class Object Instance members

- Create and use objects from class
- Class members defined without ‘**static**’
 - New memory allocation with each object created
 - Object Instance scope
 - DOES NOT EXIST UNTIL object is created with “new”
 - Java Garbage Collection (GC) automatically deletes objects when no longer needed.

jvm creates object when new is found

objects are ephemeral (lasts for a short time), garbage collector takes care of deletion of objects

Simple Class Name

```
public class Name {  
    // state is one String  
    public String n = "Dan";  
}
```

- Class **Name** is a container class for a String
 - See class **Java.Lang.String** <-- fully specified name
- Class **Name** is a Reference Type
- Object instance Member data is a String named ‘n’ holding a String value

Use Simple Class Name

```
// create object on heap and assign reference to obj  
Name obj = new Name();  
  
// use object on heap through reference in obj  
System.out.println(obj.n);      // show #1 init state  
obj.n = "Daniel";              // overwrite state  
System.out.println(obj.n);      // #1 current state  
Name obj2 = new Name();         // create object #2  
System.out.println(obj2.n);     // show #2 init state  
System.out.println(obj.n);      // #1 current state
```

Use Simple Class Name

CONSOLE OUTPUT

Dan

Daniel

Dan

Daniel

Simple Class Label

```
public class Label {  
    // state is one String  
    public static String n = "Dan";  
}
```

- Class **Label** is a container class for a String
 - See class **Java.Lang.String**
 - Class **Label** is a Reference Type
 - Static class Member data is a String named ‘n’ holding a String value
-

Use Simple Class Label

```
// use class Label  
  
System.out.println(Label.n);      // show init state  
  
Label.n = “Daniel”;              // overwrite state  
  
System.out.println(Label.n);      // show current state  
  
Label.n = “Danny”;              // overwrite state  
  
System.out.println(Label.n);      // show current state  
  
System.out.println(Label.n);      // show current state
```

Use Simple Class Name

CONSOLE OUTPUT

Dan

Daniel

Danny

Danny

Java Reference Type

- A Class is a reference type
 - To instantiate an object from a class:
 - 1.Using keyword “**new**”
 - 2.Calling a class constructor
- Must Create ALL Objects with “**new**”
 - EXCEPT String objects

Java Reference Type

- To Create a Person object:

Person dan = null;

dan = new Person();

- Data Type is “**Person**” class
- Variable Name (Identifier) is “**dan**”
- Class constructor is “**Person()**”

Java Reference Type

- To Create a Student object:

```
Student sam = new Student();
```

- Data Type is “**Student**” class
- Variable Name (Identifier) is “**sam**”
- Class constructor is “**Student()**”

Java Reference Type

- To Create a container object:

```
List<String> names = null;
```

```
names = new ArrayList<>();
```

- Data Type is “**List<String>**” interface
- Variable Name (Identifier) is “**names**”
- Class constructor is “**ArrayList<>()**”,
where <String> is compiler inferred

Java String: Java Reference Type

- Character String
 - “This is a LITERAL character string.”
- A String is a Reference Type
 - java.lang.String** class
- A String is immutable
- **NOT** an array of characters terminated by a null character (C Language).
 - A Java String object is **NOT** a C language string.

Java String

- Special String treatment:
 - Enclosing characters in double quotes **automatically** creates a String object:
`String name = "Dan";`
 - Identifier **"name"** contains a reference to a String object containing the immutable value of **"Dan"**.

Java String

- For String objects, Use of the ‘*new*’ keyword is optional (and **discouraged**)
 - Reference: <https://www.geeksforgeeks.org/interning-of-string/>
 - Java *string pool* and *string interning*.
 - Both memory (and it’s allocation time) are conserved by saving immutable strings in a pool. When a new string is created, **if it is a repeated string**, a reference to an already preserved immutable string in the pool is established in lieu of a new created string.

Java String

- Use of the ‘new’ keyword is optional (and discouraged) for creating String objects.
- DO

```
String s = “abc”; // allows interning
```

- DO NOT

```
String s = new String(“abc”); // forces new string
```

- String objects

String Interning is a method of storing only one copy of each distinct String Value, which must be immutable. Applying String.intern() on a couple of strings will ensure that all strings having the same contents share the same memory.

Array: Java Reference Type

- To Create a fixed size array container object:

```
int [] myArray = new int[3];
```

- Data Type is “**int []**” int array
 - Variable Name (Identifier) is “**myArray**”
 - The array is created for ONLY three integers by using “**int[3]**”
-

Array: Java Reference Type

“In the Java programming language, *arrays* are objects...”

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-10.html>

“An *object* is a *class instance* or an *array*. “

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.3.1>

Java Reference Type

- To Create a fixed size array container object:

```
int [] myArray = { 1, 2, 3 };
```

- Data Type is “**int []**” int array
- Variable Name (Identifier) is “**myArray**”
- The array is created for ONLY three integers by using the initializer “**{1,2,3}**”

Java Pass Primitives By Value

- Primitive data types are int, double, etc.
- Memory for Primitive data types are allocated on the stack
- Copies of Primitive data types are passed to methods
- Methods CAN NOT modify the Original primitive data type.

Java Pass Object Reference By Value

- Objects are Reference Types
 - References point to Object allocation in heap memory
 - TWO memory allocations are needed to use an object.
 1. Object allocated on the heap
 2. Reference (pointer) allocated on stack, pointing to Object allocation on the heap
 - References passed to methods are copies
 - Copies STILL POINT TO SAME OBJECT
-

Simple Class N

```
public class N {  
    public int n = 0; // state is one int  
}
```

- Class N is a container class for an integer
 - See class **Java.Lang.Integer**
 - Class N is a Reference Type
 - Object instance Member data is an integer named ‘n’ holding an integer value
-

sillySwap method

```
public void sillySwap(No1, No2) {  
    N temp = o1;      // save for later  
  
    System.out.println("Swap object references:");  
    o1 = o2;  
    o2 = temp;        // original o1  
    // COPIES of references have changed  
}
```

showObjects method

```
// output the state of each object on console  
public static void showObjects(No1, No2)  
{  
    System.out.println(" " + o1.n + " " + o2.n);  
}
```

Use SillySwap method

```
public void sillySwapObjects() {  
    N o1 = new N();      // create object 1  
    N o2 = new N();      // create object 2  
    o1.n = 1; // set value 1 in object 1  
    o2.n = 2; // set value 2 in object 2  
    ValueN.showObjects(o1, o2); // 1 2  
    ValueN.sillySwap(o1, o2); // useless swap  
    ValueN.showObjects(o1, o2); // 1 2  
}
```

Use SillySwap method

Swap object references produces:

Console Output:

1 2

1 2

smartSwap method

```
public void smartSwap(N o1, N o2) {  
    N temp = new N();  
    temp.n = o1.n    // save for later  
  
    System.out.println("Swap object state:");  
    o1.n = o2.n;  
    o2.n = temp.n;  // original o1 state  
    // state of Objects have changed  
}
```

Use smartSwap method

```
public void smartSwapObjects() {  
    N o1 = new N();      // create object 1  
    N o2 = new N();      // create object 2  
    o1.n = 1; // set value 1 in object 1  
    o2.n = 2; // set value 2 in object 2  
    ValueN.showObjects(o1, o2); // 1 2  
    ValueN.smartSwap(o1, o2); // swap state  
    ValueN.showObjects(o1, o2); // 2 1  
}
```

Use SmartSwap method

Swap object state produces:

Console Output:

1 2

2 1

CSYE 6200

Concepts of Object Oriented Design

Java Classes and Objects

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Java Classes and Objects
 - 1. Java Object Oriented Programming
 - 2. Package organization of java classes
 - 3. Java Class Details

Java Object Oriented Programming

- Object Oriented Programming (OOP)
 1. Classes are specified
 2. Classes are instantiated into useable objects
 3. Objects are used as the functional building blocks of the executing java program

Person Class

```
package edu.neu.cyse6200;  
public class Person { // fully specified name: edu.neu.cyse6200.Person  
    public int age = 0;  
    public String name = null;  
    // class default constructor  
    public Person() {  
        this.age = 3;  
        this.name = "Joe";  
    }  
}
```

NOTE: ‘public’ class data for trivial example

Driver Class specification

```
public class Driver {  
    // main() method  
    public static void main(String [] args) {  
        Person object1 = new Person();  
        System.out.println(object1.name  
            + " is " + object1.age);  
    }  
}
```

Java OOP: Class Specification

- Class specification
 - Each Java class is written (coded in java) in a single ‘*.java*’ text file
 - Java code (classes) **MUST** be compiled using the java compiler:

```
javac Driver.java Person.java
```

Java OOP: Object Instantiation

- Object Instantiation
 - A compiled Java program is executed (begins running) using the ‘**java**’ command:

java *Driver*

- program execution ALWAYS begins in **main()** method
- A running java program executes java statements one after another beginning with the first java statement in (the designated) **main()** method

Java OOP: Object Instantiation

- Classes are instantiated into useable objects
 - Program execution begins in **main()** method in the Driver class
 - Java program executes **java statements** one after another beginning in the **main()** method.
 - ALL java statements **end with a semicolon ‘;’**
 - Java code in **main()** method will:
 - Instantiate objects from class specifications
 - Use objects for ALL program execution
 - Program execution ends when **main()** method exits

Java OOP: Object Instantiation

- Program execution uses program variables
- Program variables are named memory locations used to contain data for program execution
- All program variables MUST BE DECLARED (both type and name announced to java compiler) before they can be used by java program

Java OOP: Object Instantiation

- The **main()** method in class **Driver**:

```
public class Driver {  
    public static void main(String [] args) {  
        Person object1 = new Person();  
  
        ...  
    }  
}
```

Java OOP: Object Instantiation

1. Declare program reference variable ‘**object1**’:

Person object1;

- Type: class **Person**
- Name: **object1**

2. Instantiate **Person** object from class using keyword ‘*new*’ AND **Person** class constructor

new Person();

3. Assign (write/save) Instantiated **Person** object to (memory location named) ‘**object1**’

- ALL DONE IN A SINGLE Java STATEMENT

Person object1 = new Person();

Java OOP: Object Usage

- The **main()** method in class Driver:

```
public class Driver {  
    public static void main(String [] args) {  
        ...  
        System.out.println(object1.name  
                           + “ is “ + object1.age);  
    }  
}
```

Java OOP: Object Usage

- Objects are used as building blocks for a java program
- With one exception, we must instantiate java class as an object to use its members.
 - A class' **static** members may be used without instantiation as an object.
- Use the ‘.’ (dot) to access public data and methods in static classes and instantiated objects

Java OOP: Object Instantiation

```
System.out.println(object1.name  
+ " is " + object1.age);
```

- The **main()** method uses various objects to access member data, *e.g.*, *out*, *name*, *age*, and methods, *e.g.*, *println()*
- Java code use the ‘.’ (dot) syntax to access (static or object instantiated) members defined in a class

Java OOP: Object Instantiation

```
System.out.println(object1.name
+ “ is “ + object1.age);
```

- Use the Dot “.” to :
 1. Access the class **System** static ‘**out**’ object
 2. Call the ‘**out**’ object’s ‘**println()**’ method
 3. Use the (instantiated class **Person**) ‘**object1**’ to access its ‘**age**’ and ‘**name**’ data members
 - ALL IN A SINGLE JAVA STATEMENT

Person Class usage

```
public class Driver{  
    public static void main(String[] args) {  
  
        // instantiate object from class using ‘new’ and a  
        // class Person constructor  
  
        Person obj = new Person();  
  
        System.out.println(obj.name  
            + “ is “ + obj.age + “ years old.”);  
    }  
}
```

Person Class usage

OUTPUT

Joe is 3 years old.

Package

- Package
 - Organization of Java class libraries
 - Class libraries are related
 - Hierarchical dot'.' separated name

Package (cont'd)

- Package Name Convention
 - All lower-case package name begin with top level domain
 - edu, com, org, mil, ca, de, uk
 - Followed by organization name
 - ibm, neu, mit, microsoft
 - Followed by any groups, projects or sub-projects within the organization ex: [cyse6200](#)

// Class Person: fully specified name: [edu.neu.cyse6200.Person](#)

Package (cont'd)

- Package name examples
 - java.lang
 - java.util
 - java.awt
 - java.swing
 - edu.neu.csye6200.lecture1.misc
- [edu.neu.csye6200.Person](#)

Class

- Class

```
public class MyName [ extends MySuperClass ] [  
    implements MyInterface ] {
```

- Data
- Constructor
- Method
- }

- All outer class definitions MUST BE public
 - Inner class (defined in a class) may be private

Class (cont'd)

- Class
 - Concrete
 - Declared
 - Fully implemented methods
 - Can be instantiated to create objects

Class (cont'd)

- Class
 - Abstract
 - Declared: public, private or protected members
 - Partially implemented
 - Contains one or more abstract methods
 - Data
 - final or non final
 - static or non-static
 - Must be extended (keyword extends)
 - Cannot be instantiated (without completing implementation)

Class (cont'd)

- Class
 - Interface
 - Contains ONLY (implicitly) public methods
 - Implicitly abstract methods are unimplemented
 - Java 8 ‘default’ Methods are implemented
 - Java 8 ‘static’ Methods are implemented
 - Data
 - static (class variables) ONLY
 - final (immutable constant values) ONLY
 - Must be implemented (keyword implements)
 - Cannot be instantiated (without completing implementation)

Class (cont'd)

- Class
 - Data
 - Attribute
 - Field
 - Constructor
 - One or more special Method to instantiate and initialize objects
 - Method
 - Function
 - Operation
 - Behavior
-

Class (cont'd)

- Class
 - Data declaration
 - [static] [final] [public | protected | private] type name
 - [= initializer] ;

Data Types

- Primitive types
 - byte, short, int, long, float, double, boolean, char
 - Passed by value
 - Typically, Stack memory allocation

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Data Types

- Reference types
 - Class
 - Passed by reference
 - Heap memory allocation
 - Automatic Garbage Collection (GC)
 - NEVER TO FREE (C++ delete) heap allocation

Class (cont'd)

- Class
 - Static: class global data
 - Single instance of data
 - Associated with class
 - Object instantiation not required
 - Program scope
 - Static: method, program scope
 - Can not be overridden
 - Should always be accessed in ‘static’ way, i.e., with class name and not with an object reference
 - `System.out.println()` (`out` is a *static* member of `System`)

Class (cont'd)

- The following are NEVER ‘**static**’ in Java
 - Constructor
 - Allowed in other Object-Oriented languages to initialize static data members
 - Java instead uses static initialization block

```
public class Driver {  
    static {  
        ...  
    }  
}
```
 - Outer class
 - Inner class may be a ‘**static**’ member of outer class

Class (cont'd)

- Class
 - Non-static: object instance data
 - Default
 - Independent instance with each object created
 - Object instantiation required
 - Heap memory allocation
 - Object Reference assigned to variable
 - Reference points to Object in heap memory

Class (cont'd)

- Class
 - Non-static: object instance method
 - Default
 - Object instantiation (*new*) IS required
 - Does not exist until object is created
 - Object Reference assigned to variable
 - Method is called using object reference
 - » double price = new Item().getPrice();
 - ‘this’ in object instance method is reference to current object on heap
 - Reference points to Object in heap memory

Class (cont'd)

- Class
 - **final**
 - Immutable data item: Constant data (init once)
 - Independent constant with each object
 - » final int JOB_ID = 347;
 - » Final String LABEL = “EMPLOYEE”;
 - Single instance of data
 - » static final int ERROR_CODE = 147;
 - » static final String ERROR= “Invalid Input Parameter”
 - Immutable method
 - Cannot be overridden by inheritance (like C++ non-virtual method)
 - NEVER use ‘**final**’ for a class constructor

Class (cont'd)

- Class
 - Access Modifiers
 - Public
 - Protected
 - (Default)
 - Private
 - Provides Data Hiding (Abstraction)
 - Applicable individually to each member of class

Final Classes: When you declare a class as final, it means that the class cannot be extended (i.e., you cannot create subclasses). This is often done to prevent the class from being modified or extended by other developers. Constructors are not meant to be inherited, so making a constructor final would not provide any additional benefit in terms of preventing inheritance.

Constructor Inheritance: Constructors are not inherited by subclasses in Java. Subclasses do not inherit constructors from their superclasses. They have their own constructors, which can call the constructors of their superclasses using super(). Since constructors are not inherited, making them final is unnecessary.

Object Initialization: Constructors are used for object initialization, and making them final would defeat their primary purpose. The final keyword is typically used to indicate that something cannot be changed or overridden, but constructors need to be called to create objects. If a constructor were marked as final, you wouldn't be able to create objects of that class, which would be impractical.

Class (cont'd)

- Class
 - Access Modifiers
 - Public
 - All access
 - » Accessible by classes within package
 - » Accessible by sub-class
 - » Accessible by classes outside package

Access Modifier	Classes within Package	By Subclass	Outside Package
public	Yes	Yes	Yes
protected	Yes	Yes	No
default	Yes	No	No
private	No	No	No

public - has all access

private - only class itself has access to its private members

Class (cont'd)

- Class
 - Access Modifiers
 - Protected
 - Class, Package **and Sub-class** access
 - » Accessible by classes within package
 - » **Accessible by sub-class**
 - » NOT Accessible by classes outside package

Class (cont'd)

- Class
 - Access Modifiers
 - Default: Neither Public, Protected nor Private
 - Package Private
 - » Accessible by classes within package
 - » **NOT Accessible by sub-class**
 - » NOT Accessible by classes outside package

Class (cont'd)

- Class
 - Access Modifiers
 - Private
 - Class private
 - » **ONLY Class itself has access to private members**
 - » NOT Accessible by classes within package
 - » NOT Accessible by sub-class
 - » NOT Accessible by classes outside package

Class (cont'd)

- Class
 - Constructor
 - Special Method used to instantiate objects
 - Constructor NAME is IDENTICAL to class name
 - MUST NOT specify a return value type OR void
 - Default Constructor
 - No arguments
 - Compiler provided IF NO CONSTRUCTORS
 - Multiple Constructors
 - Overloaded
 - » Different signatures (i.e., number and types of args)
 - Provides **Static Polymorphism**

Static Polymorphism		Dynamic Polymorphism
Overloading		Overriding (@Override)
Same method name, diff parameters		Same method name, but in subclass
Compile Time		Run time

Class (cont'd)

- Class
 - Constructor
 - Never ‘**static**’ in Java
 - Allowed in C#, a *static constructor* would initialize *static* data members: Java uses static initialization block instead
 - Never ‘**final**’
 - **Only Allowable Modifiers for constructor:**
 - **public**
 - **private**
 - **protected**

```
public class MyClass {  
    static int staticField1;  
    static String staticField2;  
  
    // Static initialization block  
    static {  
        staticField1 = 42;  
        staticField2 = "Hello, world!";  
    }  
}
```

In Java, a static initialization block, also known as a static initializer or a static initialization block, is a code block enclosed in braces and preceded by the static keyword. It is used to initialize static members of a class. Static initialization blocks are executed when the class is first loaded by the classloader, and they are executed only once. This makes them useful for performing one-time setup for static fields.

Class (cont'd)

- Class
 - Method
 - Also called function, operations, behaviors
 - Abstract: declaration only: no implementation
 - Concrete: declaration and implementation
 - MUST specify a return value type OR void
 - Overloaded Methods
 - Same names
 - Different signatures (i.e., number and types of arguments)
 - Different return types DOES NOT distinguish methods
 - Provides **Static Polymorphism**
 - Override (@Override) **run-time Polymorphism**

Java Class Summary

- Class Summary
 - Single Outer class in **.java** source code file
 - Name of class is ALSO used for **.java** file name
 - Outer class MUST ALWAYS ‘**public**’
 - Outer class MUST NEVER BE ‘**static**’
 - Inner class is a data member of outer class
 - Allowed to be ‘**private**’, ‘**protected**’ or ‘**public**’
 - Allowed to be ‘**static**’

Java Class Summary

- Class Summary
 - Concrete
 - Fully implemented methods
 - Abstract
 - Contains one or more abstract methods
 - *CANNOT be instantiated - MUST be extended
- Interface
 - Contains public abstract, default and static methods
 - *CANNOT be instantiated - MUST be implemented
 - * NOTE: UNLESS implementation is completed

Benefits

- Java class Benefits
 - Encapsulation
 - Data and Method associated together in class
 - Abstraction
 - Data hiding
 - Access Modifiers
 - » Public, Private, Protected
 - Functionality hiding
 - Abstract method as API
 - Interface as API

CSYE 6200

Concepts of Object Oriented Design

Java Enumerated Types

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Enum

Enum

java.lang.Enum

public enum Color {RED, WHITE, BLUE};

- Trailing semicolon is OPTIONAL
public enum Color {RED,WHITE, BLUE}
- Explicitly listed set of strongly typed constants
- Immutable: implicitly final static: Cannot be changed once initialized

Enum is a immutable java class - type class

Enum

java.lang.Enum

```
public enum Explosion {GUNSHOT,  
GRENADE, ABOMB};
```

- Explicitly listed set of strongly typed constants
- Immutable: *implicitly final static*: Cannot be changed once initialized

Enum

- Declared enum “stockName” can only be assigned enum type from declared set of enum constant values, “EBAY”, “IBM”, “GOOGLE”, “YAHOO”, “ATT”

```
public enum Stock{  
    EBAY, IBM, GOOGLE, YAHOO, ATT  
}
```

```
Stock stockName = Stock.EBAY;  
stockName = 1; // Compilation Error
```

Enum

- Declared as class data member
- Prefer over integer or string codes to increase compile-time checking
- Can have constructors, methods and variables

Enum

- Constructors:
 - Private and never invoked directly in code
 - Called *automatically* when enum is initialized
- Declared inside a class but not in a method
- Declared public so ok to use outside a class
 - BUT Cannot be declared final
- Prefer over integer or string codes to increase compile-time checking

Enum Example

```
public enum Fruit{  
    APPLE("A") , KIWI("K"), GRAPE("G"), PEAR("P");  
  
    private String fruitLetter;  
  
    private Fruit(String s) {  
        fruitLetter = s;  
    }  
    public String getFruitLetter() {  
        return fruitLetter;  
    }  
}
```

Enum Usage

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

```
    System.out.println("'"kiwi' enum constant is: "
+ Fruit.KIWI.getFruitLetter());
}
```

OUTPUT:

```
'kiwi' enum constant is: Value: K
```

Example 2: Enum Planet

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6), VENUS (4.869e+24, 6.0518e6), EARTH (5.976e+24, 6.37814e6), MARS  
(6.421e+23, 3.3972e6), JUPITER (1.9e+27, 7.1492e7), SATURN (5.688e+26, 6.0268e7), URANUS (8.686e+25,  
2.5559e7), NEPTUNE (1.024e+26, 2.4746e7);  
  
    private final double mass; // in kilograms  
    private final double radius; // in meters  
    Planet(double mass, double radius) { this.mass = mass; this.radius = radius; }  
    private double mass() { return mass; }  
    private double radius() { return radius; }  
    // universal gravitational constant (m3 kg-1 s-2)  
    public static final double G = 6.67300E-11;  
    double surfaceGravity() { return G * mass / (radius * radius); }  
    double surfaceWeight(double otherMass) { return otherMass * surfaceGravity(); }  
    public static void main(String[] args) {  
        if (args.length != 1) {  
            System.err.println("Usage: java Planet <earth_weight>");  
            System.exit(-1);  
        } // end if  
        double earthWeight = Double.parseDouble(args[0]);  
        double mass = earthWeight/EARTH.surfaceGravity();  
        for (Planet p : Planet.values()) System.out.printf("Your weight on %s is %f\n", p, p.surfaceWeight(mass));  
    } // end main  
} // end enum Planet
```

Enum Planet

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6),  
    VENUS (4.869e+24, 6.0518e6),  
    EARTH (5.976e+24, 6.37814e6),  
    MARS (6.421e+23, 3.3972e6),  
    JUPITER (1.9e+27, 7.1492e7),  
    SATURN (5.688e+26, 6.0268e7),  
    URANUS (8.686e+25, 2.5559e7),  
    NEPTUNE (1.024e+26, 2.4746e7);  
};
```

.....

Enum Planet (cont'd)

.....

```
private final double mass; // in kilograms  
private final double radius; // in meters
```

```
Planet(double mass, double radius) {  
    this.mass = mass; this.radius = radius;  
}  
private double mass() { return mass; }  
private double radius() { return radius; }
```

.....

Enum Planet (cont'd)

.....

```
// universal gravitational constant (m3 kg-1 s-2)
public static final double G = 6.67300E-11;

double surfaceGravity() {
    return G * mass / (radius * radius);
}

double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
}
```

.....

Enum Planet (cont'd)

.....

```
public static void main(String[] args) {  
    if (args.length != 1) {  
        System.err.println("Usage: java Planet  
<earth_weight>");  
        System.exit(-1);  
    } // end if  
    double earthWeight =  
    Double.parseDouble(args[0]);  
    double mass =  
    earthWeight/EARTH.surfaceGravity();  
    .....
```

Enum Planet (cont'd)

.....

```
for (Planet p : Planet.values())
    System.out.printf("Your weight on %s is %f
%n", p, p.surfaceWeight(mass));

} // end main
} // end enum Planet
```

CSYE 6200

Concepts of Object Oriented Design

String class

Daniel Peters

d.peters@neu.edu

-
- Lecture:
 1. Java String class
 2. Java StringBuilder class

Java String

- Character String
 - “This is a LITERAL character string.”
- A String is a Reference Type
 - java.lang.String** class
- A String is immutable
- **NOT** an array of characters terminated by a null character (C Language).
 - A Java String object is **NOT** a C language string.

Java String

- Special String class treatment:
 - Enclosing characters in double quotes **automatically** creates a String object:
`String name = "Dan";`
 - Identifier **"name"** contains a *reference* to a String object containing the immutable value of **"Dan"**.

Java String

- For String objects, Use of the ‘new’ keyword is optional (and discouraged)
 - Reference: Java **String pool** and **String interning**.
 - Both memory (and its allocation time) are conserved by saving immutable Strings in a pool. When a new String is created, **if it is a repeated String**, a reference to an already preserved immutable String in the pool is established in lieu of a new String created.

Java String

- Use of the ‘new’ keyword is optional (and discouraged) for creating String objects.
- DO

```
String s = “abc”; // allows interning
```
- DO NOT

```
String s = new String(“abs”); // forces new string
```
- String objects

String Operations

- **String.toUpperCase()**
 - Converts String to ALL CAPS
- **String.toLowerCase()**
 - Convert String to ALL LOWERCASE
- Compare: **s1.compareTo(s2);**
 - Returns 0 indicating lexographically equal strings

String Split

- String split

```
String s = new String("Dan,16,4.0");
```

```
String [] tokens = s.split(",");
```

```
System.out.println("Student:"
```

```
+ " NAME: " + tokens[0]
```

```
+ ", AGE: " + tokens[1]
```

```
+ ", GPA: " + tokens[2]);
```

- OUTPUT:

```
Student: NAME: Dan, AGE: 16, GPA: 4.0
```

Integer to String Conversion

- **Integer.parseInt()**
 - Convert String to int value

```
String s = "17"; // String representation of int 17
```

```
int age = 0;
```

```
try {
```

```
    age = Integer.parseInt(s); // convert String to int
```

```
} catch (NumberFormatException e) {
```

```
    System.out.println(s + " is not a number!");
```

```
    e.printStackTrace();
```

```
}
```

```
System.out.println(s + " is Age: " + age);
```

Integer to String Conversion

- CONSOLE OUTPUT:

17 is Age: 17

Double to String Conversion

- **Double.parseDouble()**
 - Convert String to double value

```
String s = "4.0"; // String representation of 4.0
double gpa= 0;
try {
    gpa = Double.parseDouble(s); // String to double
} catch (NumberFormatException e) {
    System.out.println(s + " is not a number!");
    e.printStackTrace();
}
System.out.println(s + " is GPA: " + gpa);
```

Double to String Conversion

- CONSOLE OUTPUT:

4.0 is GPA: 4.0

SubString

- SubString

```
String s = new String ("abcd");
```

```
int ix1 = 1;
```

```
int ix2 = 3;
```

```
sub = s.substring(ix1); // bcd starts with ix1  
[ix1 , )
```

```
sub = s.substring(ix1,ix2); // bc
```

```
int ix1 = 0; starts with ix1, ends with ix2 - 1  
[ix1, ix2)
```

```
sub = s.substring(ix1,ix2); // abc
```

StringBuilder

```
StringBuilder sb = new  
StringBuilder("Peter");  
sb.append(",");  
sb.append("Paul");  
sb.append(",");  
sb.append("Mary");  
sb.append(",");  
System.out.println(sb.toString());
```

StringBuilder

• CONSOLE OUTPUT:

Peter, Paul, Mary,

concat() vs "+" Operator :

Both are equally less efficient than String builder
Use String Builder for better performance

But, Modern Java compilers can optimize +
concatenation for simple cases

Why should you choose String Builder over "+" operator?

Reason	String builder	+ Operator
Efficiency	Looping: uses same string object for concatenation	Looping: new string object is created for each concatenation
Mutability	Open to modification - concat, insert, delete on same object	New string objects are created for every concat "+"
Readability	More readable & maintainable	Unclear multiple "+" signs
Thread Safe	String-Builder is not thread safe, but it is faster than String-Buffer which is thread safe - synchronized	Not thread safe

CSYE 6200

Concepts of Object-Oriented Design

repeat

Inheritance

Daniel Peters

d.peters@neu.edu

- Lecture
 - 1. Inheritance
 - 2. Java Concrete Class
 - 3. Java Abstract Class
 - 4. Java Interface

Relationships

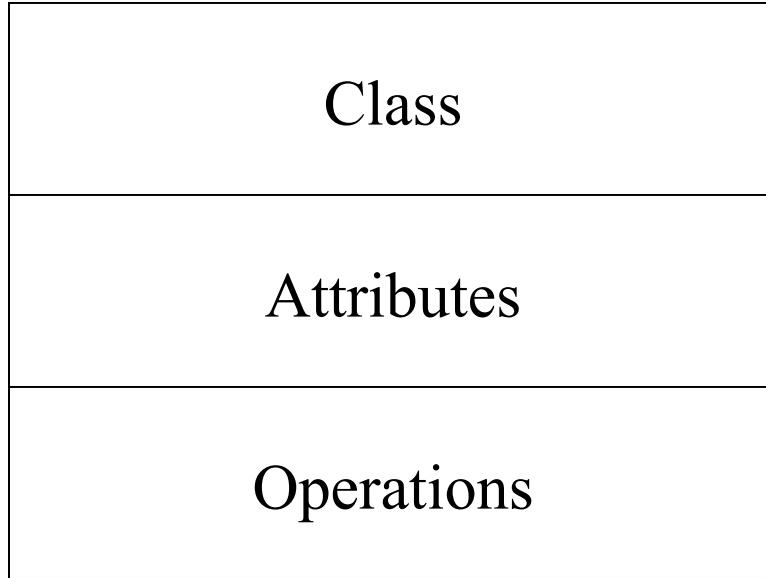
- Generalization: Inheritance Is-A Relationship
 - Child Is-A Parent

Parent



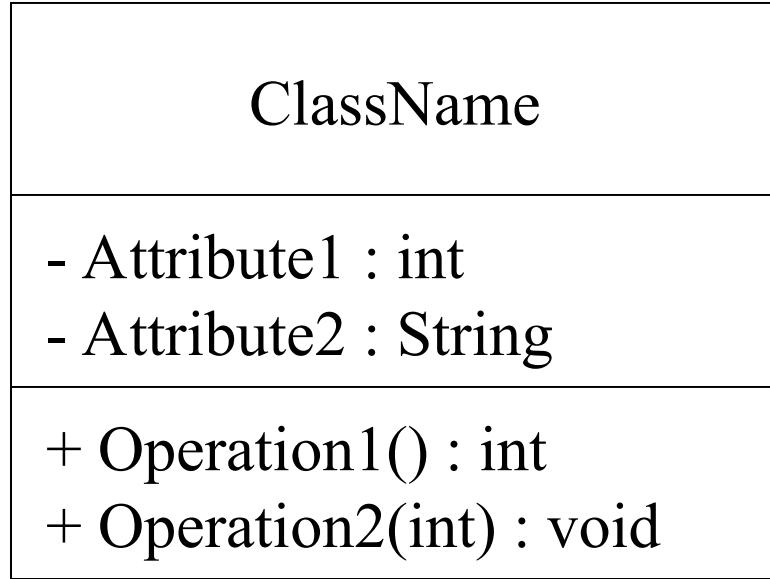
Child

UML Class Diagram

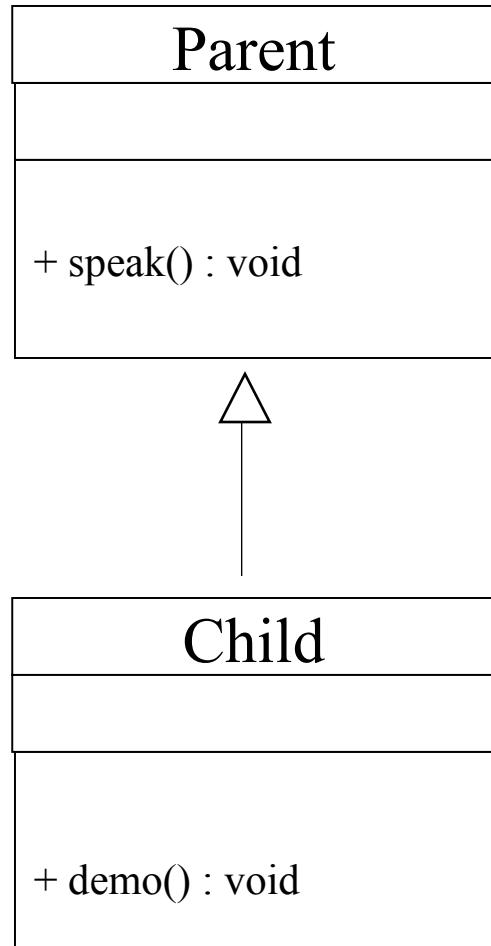


UML Class Diagram

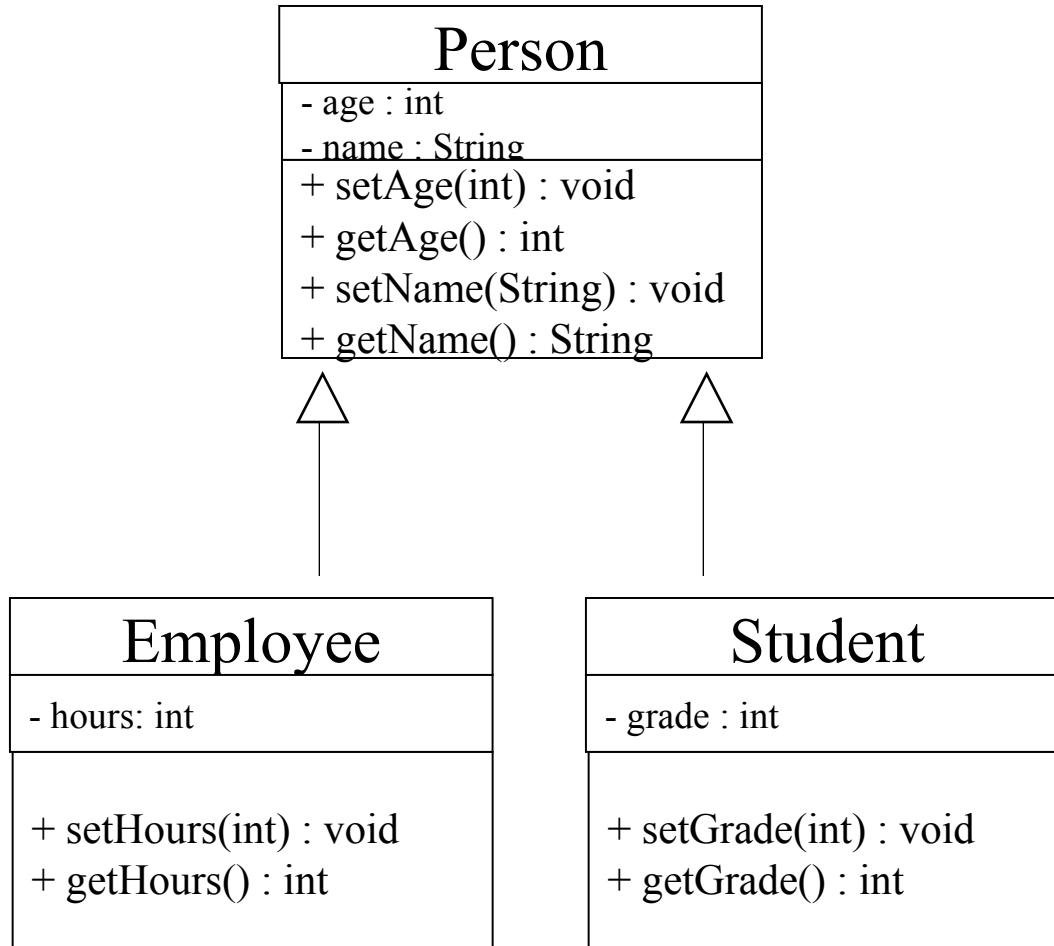
- private
+ public



Inheritance Class Diagram



Person Class Diagram



Class

- Class

```
public class MyName [ extends MySuperClass ]  
[ implements MyInterface1, MyInterface2 ] {
```

- Data
- Constructor
- Method
- }

- Java outer class:

- MUST be **public** and NEVER *static*
- Inner classes are declared inside a class

Concrete Class

- Class
 - Concrete
 - Declared: public, private or protected members
 - Fully implemented methods
 - Data
 - final or non final
 - static or non-static
 - Can be instantiated to create objects

Abstract Class

- Class
 - Abstract
 - Declared: public, private or protected members
 - Partially implemented
 - Contains one or more abstract methods
 - Data
 - final or non final
 - static or non-static
 - Must be extended (keyword extends)
 - Cannot be instantiated (without completing implementation)

Interface

- Interface
 - Contains ONLY public methods (implicit)
 - Implicitly ‘abstract’ methods are unimplemented
 - Java 8 ‘default’ Methods are implemented
 - Java 8 ‘static’ Methods are implemented
 - Data
 - static (class variables) ONLY
 - final (immutable constant values) ONLY
 - Must be implemented (keyword implements)
 - Cannot be instantiated (without completing implementation)
-

Class

- Class
 - Data
 - Attribute
 - Field
 - Constructor
 - One or more special Methods to instantiate and initialize objects
 - Method
 - Function
 - Operation
 - Behavior
-

Class (cont'd)

- Class
 - Data declaration
 - [static] [final] [public | protected | private] type name
 - [= initializer] ;

Class (cont'd)

- Class
 - Class Constructor declaration
 - [public | protected | private]
 - ClassName()
 - { constructor method body } ;
- Note: *public*, *protected* and *private* are the ONLY modifiers which may be specified for a constructor

Class (cont'd)

- Class
 - Method declaration
 - [*static*] [**final**] [public | protected | private]
 - [returnType | void] methodName()
 - { method body } ;
- A *static* class method is inherited but NO overriding support
- A **final** object method is inherited but prohibits overriding

CSYE 6200

Concepts of Object-Oriented Design

Java Interface

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Java Interface

Relationships

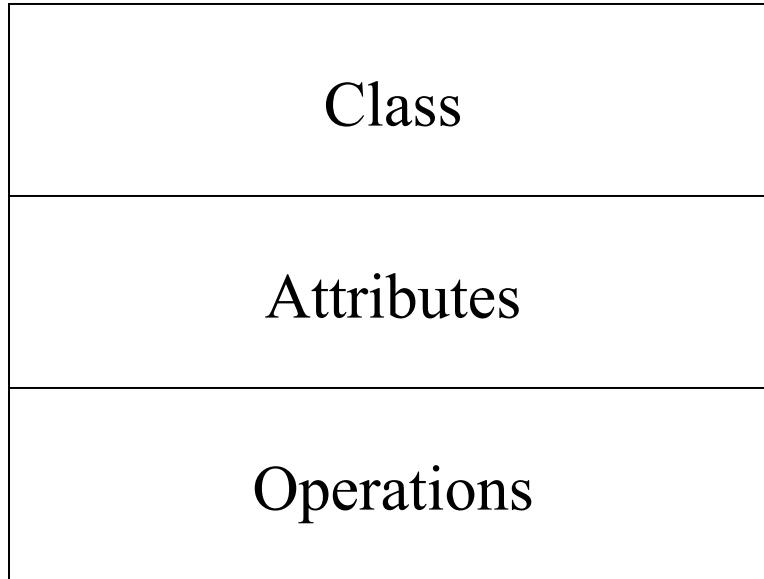
- Generalization: Inheritance Is-A Relationship
 - Child Is-A Parent

Parent



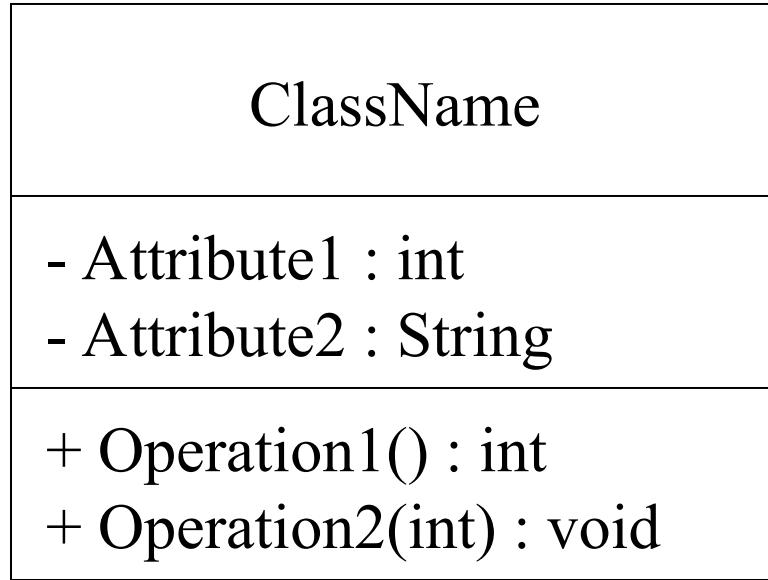
Child

UML Class Diagram

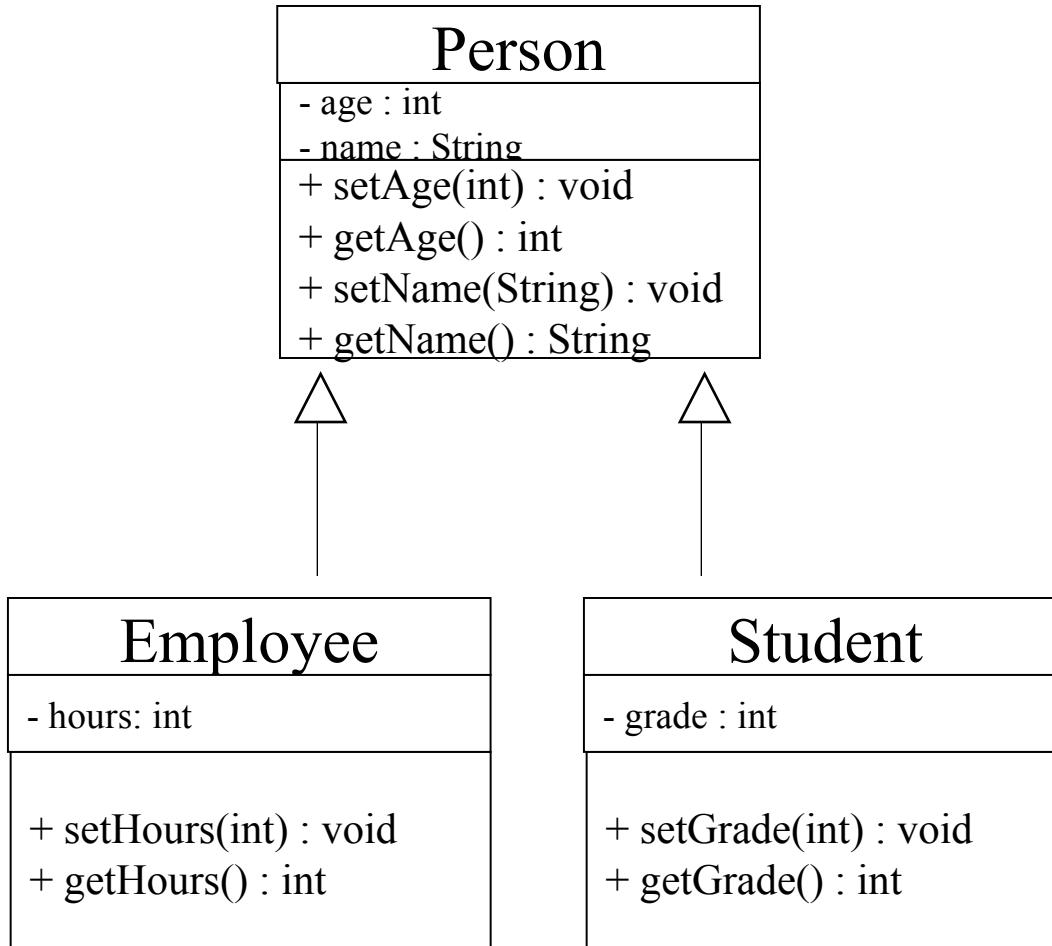


UML Class Diagram

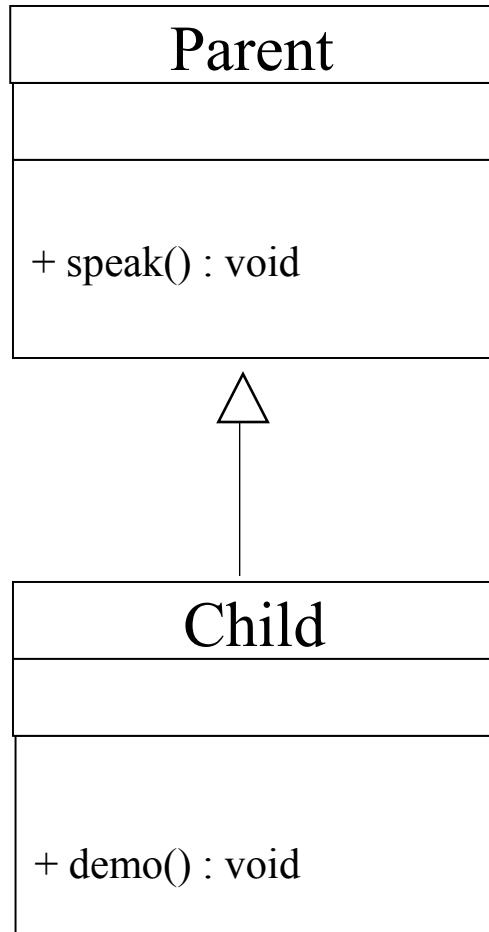
- private
+ public



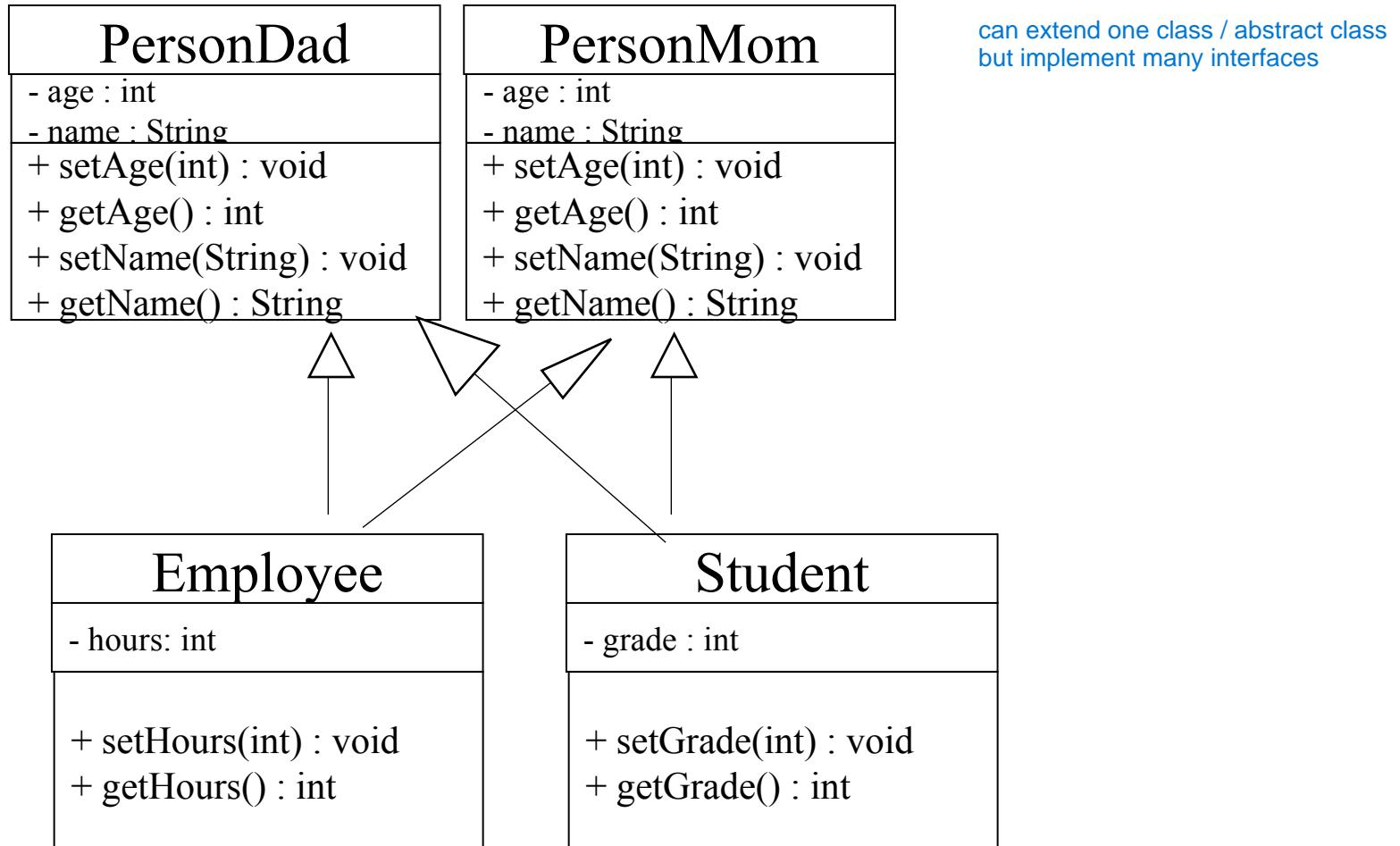
Person Class Diagram



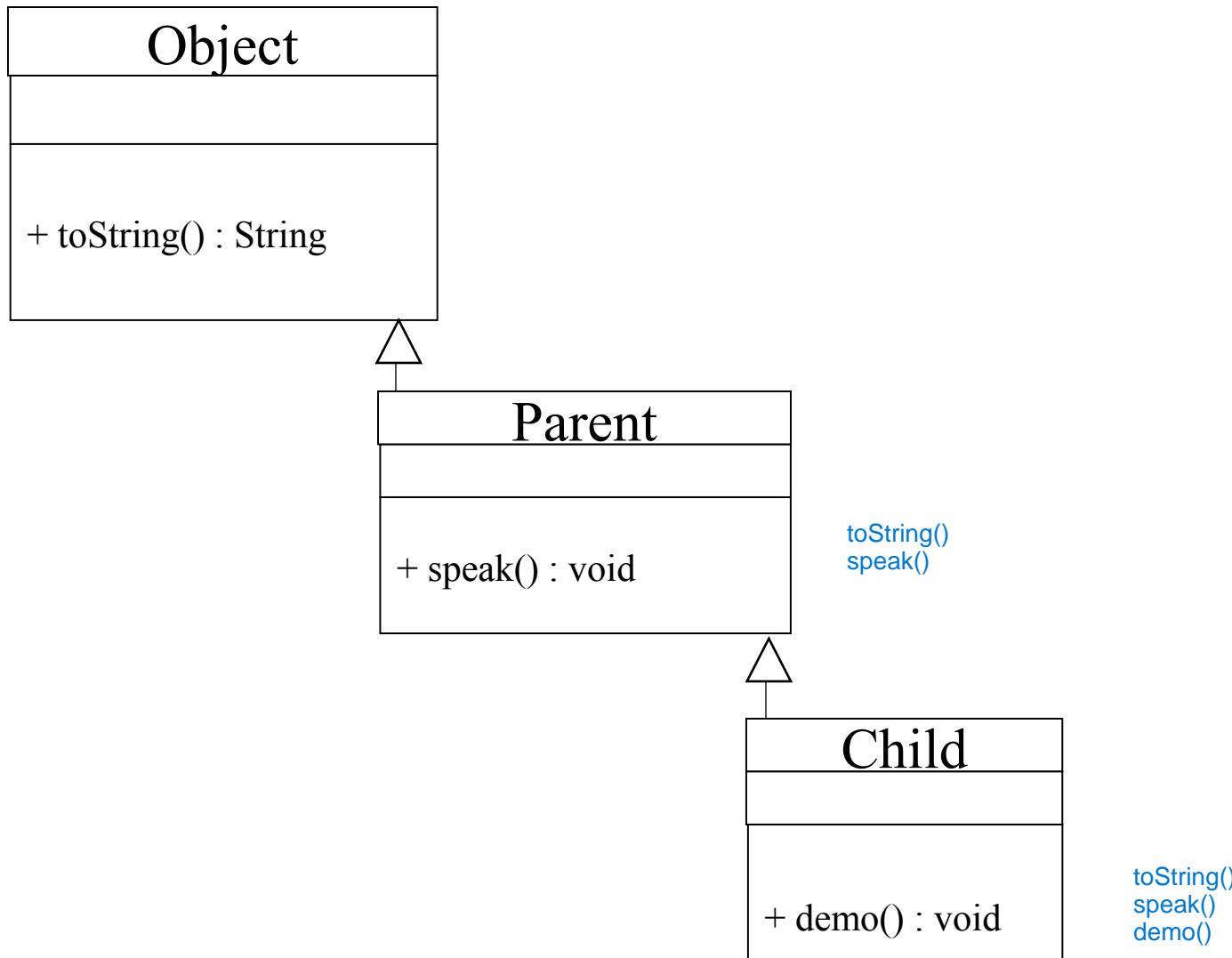
Inheritance Class Diagram



Multiple Inheritance Class Diagram



Single Inheritance Class Diagram



Class

- Class
 - class MyName [extends MySuperClass]
[implements MyInterface1, MyInterface2] {
 - Data
 - Constructor
 - Method
 - }

Concrete Class

- Class
 - Concrete
 - Declared: public, private or protected members
 - Fully implemented methods
 - Data
 - final or non final
 - static or non-static
 - Can be instantiated to create objects

Abstract Class

- Class
 - Abstract
 - Declared: public, private or protected members
 - Partially implemented
 - Contains one or more abstract methods
 - Data
 - final or non final
 - static or non-static
 - Must be extended (keyword extends)
 - Cannot be instantiated (without completing implementation)

Interface

- Contains ONLY public methods
 - ‘abstract’ methods are unimplemented
 - Java 8 ‘default’ Methods are implemented
 - Java 8 ‘static’ Methods are implemented
 - Data
 - static (class variables) ONLY
 - final (immutable constant values) ONLY
 - Must be implemented (keyword implements)
 - Cannot be instantiated (without completing implementation)
-

Java Interface

- Java Interface
 - Reference type, similar to a class
 - Contains ONLY:
 - Data
 - Public Static Final (Constants)
 - Methods:
 - ALL METHODS IMPLICITLY PUBLIC
 - » Keyword public may be omitted from methods.
 - abstract methods (no body)
 - Static methods
 - Default methods (Java 8)

Java Interface

- Java Interface
 - Public:
 - Public interface is accessible to all classes
 - Otherwise, only accessible to classes in same package
 - Keyword interface
 - Must be implemented by a class
 - Class MUST implement all interface methods
 - Can be extended by other interfaces

Java Interface

```
public interface GroupedInterface extends Interface1,  
Interface2, Interface3 {  
    // constant declarations  
    // base of natural logarithms  
    double E = 2.718282;  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

<https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html>

Java Interface

public interface GroupedInterface

- Public: interface available to all classes
- Private: interface available to package ONLY
- Keyword interface required
- Keyword extends:
 - An interface can use multiple other interfaces
- ALL methods are public
 - Keyword ‘public’ optional for all methods
 - Abstract methods are signature only
 - Default and Static methods with implementations

Java Interface

- Java Interface Use
 - Must be *implemented* by a class
 - Class MUST implement all interface methods
 - Keyword: **implements**
 - Can be *extended* by another interface
 - Keyword: **extends**

Java Interface

- Java Interface as API
- “The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. **APIs are also common in commercial software products.** Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's **API is made public (to its customers), its implementation of the API is kept as a closely guarded secret**—in fact, it **may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.**”

<https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>

Java Interface

- Java Interface Examples
 - **Comparable**
 - Implement to make your class sortable by default in natural order.
 - **Comparator**
 - Implement and use to specify a specific sort order.
 - **Runnable**
 - Implement to make your class executable on a new thread.

Comparable Interface

- Sort Pizza objects by pizza price
 - Comparable Interface provides a natural order

```
public class Pizza implements Comparable<Pizza> {  
    ...  
    public int compareTo (Pizza o) {  
        return Double.compare(this.getPrice(), o.getPrice());  
    }  
}  
List<Pizza> pizzaList = new ArrayList<>();  
Collections.sort(pizzaList);      // Comparable (natural) order
```

Comparable Interface

```
public class Student extends Person implements Comparable  
< Student > {  
    . . .  
    public int compareTo ( Student o) {  
        return Double.compare(this.getGpa(), o.getGpa());  
    }  
}  
  
List< Student > students = new ArrayList<>();  
Collections.sort(students);    // Comparable (natural) order
```

Comparator Interface

```
public class CompareByPrice implements  
Comparator<Price> {  
  
    @Override  
    public int compare(Pizza o1, Pizza o2) {  
        return Double.compare(o.getPrice(), o.getPrice());  
    }  
}  
  
List<Pizza> pizzaList = new ArrayList<>();  
pizzaList.sort(new CompareByPrice());      // Comparator  
(explicit) order
```

Comparator Interface

```
public class CGPA implements Comparator<Student> {  
  
    @Override  
    public int compare(Student o1, Student o2) {  
        return Double.compare(o1.getGpa(), o2.getGpa());  
    }  
  
    List< Student > students = new ArrayList<>();  
    students.sort(new CGPA()); // Comparator (explicit) order
```

Comparator Interface

```
public class Student extends Person implements Comparable < Student > {  
    ...  
    private static class Ranking implements Comparator<Student> {  
        @Override  
        public int compare(Student o1, Student o2) {  
            // sort by GPA High to Low  
            return Double.compare(o2.getGpa(), o1.getGpa());  
        }  
    }  
    public static void ranking(List<Student> students) {  
        students.sort(new Ranking());  
    }  
    ...  
    List< Student > students = new ArrayList<>();  
    Student.ranking(students); // Comparator (explicit) GPA order
```

Comparator Interface

```
public class Student extends Person implements Comparable < Student > {  
    ...  
    private static class Ranking implements  
Comparator<Student> {  
        @Override  
        public int compare(Student o1, Student o2) {  
            // sort by GPA High to Low  
            return Double.compare(o2.getGpa(), o1.getGpa());  
        }  
    }  
    ...  
}
```

Comparator Interface

- Class implements a static sorting method

```
public class Student extends Person implements Comparable<Student>
{
    ...
    public static void ranking(List<Student> students) {
        students.sort(new Ranking());
    }
}
```

Comparator Interface

- Use the static class sorting method

```
List< Student > students = new ArrayList<>();  
Student.ranking(students);      // Comparator GPA order
```

CSYE 6200

Concepts of Object-Oriented Design

Java Polymorphism

Daniel Peters

d.peters@neu.edu

Java Class

- Class implementation in java source file (.java)
- File name identical to class name

Java Class (Cont'd)

- Implementation for one class only
 - Data members
 - Class Constructors
 - Method members

Java Class (Cont'd)

- Implementation for one class only
 - Data members
 - Class Storage specifier
 - static: Class global variable
 - Object instance variable (default)
 - Access modifier (public, private, protected, none)
 - Type
 - Name
 - Optional initializer
 - = 1;
 - = “I am an animal.”;

Java Class (Cont'd)

- Implementation for one class only
 - Class Constructors
 - Special Methods
 - Constructor name Identical to class name
 - No return type (not even ‘void’)
 - Otherwise, like methods
 - Method members
 - Methods
 - Return type (or void)
 - Method name
 - Parameters and Parameter types
 - Function body (curly braces)

OO Language comparison

- Polymorphism as implemented in Object Oriented Languages
- **Java** method override
 - Allowed by default
 - Disallowed explicitly with keyword **final**
- **C++ and C#** method override:
 - Disallowed by default
 - Allowed explicitly with keyword **virtual**

Polymorphism

- Polymorphism: Existing in many forms.
- Provides flexibility
- Multiple implementations
- Extensibility
- Polymorphism
 - Static Compile-Time Polymorphism
 - Function Overloading
 - Dynamic Run-Time Polymorphism
 - Function Overriding

Static Compile-Time Polymorphism

- Polymorphism: Existing in many forms.
- Implemented in Class Methods:
- Function Overloading:
 - Multiple methods:
 - In SAME CLASS
 - Share SAME NAME
 - DIFFERENT function signatures
 - Type and number of parameters
 - Return value type NOT part of signature

Static Polymorphism | Dynamic Polymorphism

Overloading		Overriding (@Override)
Same method name, diff parameters		Same method name, but in subclass
diff function signature		same function signature
Compile Time		Run time

Dynamic Run-Time Polymorphism

- Polymorphism: Existing in many forms.
 - Implemented in Class Methods:
 - Function Overriding:
 - Inheritance Required
 - Overridable method NOT disabled by Parent superclass (i.e. NOT final)
 - Overriding method implemented in Deriving Child subclass with IDENTICAL SIGNATURE
 - MUST USE Derived Child subclass object through Parent superclass type (API).
- Why function overriding is disadvantageous
Virtual tables - more number of lookups
- Code in hexa-decimal - code executes for very fast

Animal.java

```
package edu.neu.csye6200;

public class Animal {
    public Animal() {
    }
    public void speak() {
        System.out.println("I am an animal.");
    }
}
```

- Concrete class as Superclass and API
 - Specifies What (signature) and How (implementation)
-

Animal.java

- Specifies WHAT and HOW
 - What: method signature
 - How: method implementation
- Default implementation provided
- Every derived class **can** provide its own implementation for API speak() method
 - Animal speak() implementation inherited if not provided (overridden) by derived class.

AnimalForced.java

```
package edu.neu.csye6200;

public class AnimalForced {
    public Animal() {
    }
    public void final speak() {
        System.out.println("I am an animal.");
    }
}
```

- Concrete class as Superclass and API
 - Specifies What (signature) and How (implementation)
-

AnimalForced.java

- Specifies WHAT and HOW
 - What: method signature
 - How: method implementation
- ONLY Implementation for speak()
- Every derived class **can NOT** provide it's own implementation for API speak()
method since its final
 - AnimalForced speak() implementation is forced and can not be overriden by derived class.

AbstractAnimal.java

```
package edu.neu.csye6200;

public abstract class AbstractAnimal {
    public Animal() {
    }
    public abstract void speak(); // abstract method as API
}
```

- Abstract class as Superclass and API
- Specifies WHAT without HOW
 - What: method signature
 - How: method implementation

AbstractAnimal.java

- Specifies WHAT without HOW
 - What: method signature
 - How: method implementation
- No default implementation provided
- Every derived class must provide it's own implementation for API speak() method
 - Compiler error if not provided

Animal.java

```
public void speak() {  
    System.out.println("I am an animal.");  
}
```

- speak method in class Animal
 - speak()
- Provides implementation in function body (curly braces)

Animal.java

```
public void speak() {  
    System.out.println("I am an animal.");  
}
```

- Java allows a **non-final** method in the superclass to be overridden by a subclass that implements same identical method signature.

Dog.java

```
package edu.neu.csye6200;

public class Dog extends Animal {
    public Dog() {
    }
    @Override
    public void speak() {
        System.out.println("I am a dog.");
    }
}
```

Cat.java

```
package edu.neu.csye6200;

public class Cat extends Animal {
    public Cat() {
    }
    @Override
    public void speak() {
        System.out.println("I am a cat.");
    }
}
```

Fish.java

```
package edu.neu.csye6200;

public class Fish extends Animal {
    public Fish() {
    }
    @Override
    public void speak() {
        System.out.println("I am a Fish.");
    }
}
```

@Override

- `@Override` annotation
- NOT part of Java code
- NOT required
- VERY HELPFUL
- ALWAYS RECOMMENDED
 - Informs Java compiler of the intent of your code
 - To override Superclass method of same signature (name, arg types)
 - Allows java compiler to double-check your code
 - Compiler error if `@Override` method DOES NOT MATCH a Superclass method.

Driver.java

```
package edu.neu.csye6200;
import java.util.ArrayList;
import java.util.List;
public class Driver {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        animal.speak();      I am an animal
        dog.speak();        I am a dog
        List<Animal> list = new ArrayList<Animal>();
        list.add(animal);
        list.add(dog);
        for (Animal obj : list) { obj.speak(); }      I am an animal
                                                    I am a dog
    }
}
```

Driver.java

```
package edu.neu.csye6200;
import java.util.ArrayList;
import java.util.List;
public class Driver{
    public static void main(String[] args) {
        Animal animal= new Animal();
        Dog dog= new Dog();
        animal.speak();
        dog.speak();
        List<Animal> list = new ArrayList<Animal>();
        list.add(animal);
        list.add(dog);
        for (Animal obj : list) { obj.speak(); }
    }
}
```

Driver.java

```
// Create Animal object  
Animal animal = new Animal();  
  
// Create Dog object: Dog IS-A Animal  
Dog dog = new Dog(); // use as Dog  
  
animal.speak();           // Animal speak  
dog.speak();              // Dog speak
```

Driver.java

```
package edu.neu.csye6200;
import java.util.ArrayList;
import java.util.List;
public class Driver{
    public static void main(String[] args) {
        Animal animal= new Animal();
        Animal dog= new Dog(); // use as Animal ?
        animal.speak();
        dog.speak();
        List<Animal> list = new ArrayList<Animal>();
        list.add(animal);
        list.add(dog);
        for (Animal obj : list) { obj.speak(); }
    }
}
```

Driver.java

```
// Create Animal object  
Animal animal = new Animal();  
  
// Create Dog object: Dog IS-A Animal  
// Alternatively, use dog as type Animal  
Animal dog = new Dog(); // use as Animal  
animal.speak();           // Animal speak  
dog.speak();              // Dog speak
```

Driver.java

```
package edu.neu.csye6200;
import java.util.ArrayList;
import java.util.List;
public class Driver{
    public static void main(String[] args) {
        Animal animal= new Animal();
        Animal dog= new Dog(); // use Dog as Animal
        animal.speak();          // I am an animal
        dog.speak();            // I am a dog
        List<Animal> list = new ArrayList<Animal>();
        list.add(animal);
        list.add(dog);
        for (Animal obj : list) {
            obj.speak(); // I am an animal, ... I am a dog
        }
    }
}
```

Driver.java

```
List<Animal> list =  
    new ArrayList<Animal>();  
list.add(animal); // add obj to container  
list.add(dog);   // add obj to container  
  
// for each item in container, execute speak  
for (Animal obj : list) {  
    obj.speak();      // speak  
}
```

DriverProject.cpp

- CONSOLE OUTPUT

DriverProject main() ...

I am an animal.

I am a dog.

I am an animal.

I am a dog.

Using Polymorphism

- Driver instantiates both Animal and Dog objects:

```
Animal animal = new Animal();
```

```
Dog dog = new Dog();
```

- Dog class is derived from Animal class so any dog object *IS-A* Animal object and can be used as an Animal type without a java type mismatch error.

Using Polymorphism

- Driver uses the **Animal** object to speak:
`animal.speak(); // Animal speak`
- Driver uses the **Dog** object to speak:
`dog.speak(); // Dog speak`
- This is conventional use of an objects, no run-time polymorphism here.

Driver.java

```
List<Animal> list = new ArrayList<>();  
list.add(animal); // add obj to container  
list.add(dog);   // add obj to container
```

Using Polymorphism

- Driver declares an **Animal** container;

```
List<Animal> list = new ArrayList<>();
```

- Can contain **Animal** object

- Can contain **Dog** object

- Dog *IS-A* Animal

- Driver adds both Animal and Dog objects to **Animal** container:

```
list.add(animal);
```

```
list.add(dog);
```

Using Polymorphism

- Driver adds BOTH the Animal object AND the Dog object to the Animal container because Dog *IS-A* Animal:

```
list.add(animal);
```

```
list.add(dog);
```

Driver.java

```
// for each item in container, execute speak  
for (Animal obj : list) {  
    obj.speak(); // speak  
}
```

- obj reference variable is type **Animal**
 - Range based For Loop uses variable obj
 - obj uses **Animal** API *speak()* method

Using Polymorphism

- Driver uses each object out of the Animal container as if it were an **Animal** object because Dog *IS-A* Animal:

```
obj.speak();      // speak
```

Using Polymorphism

```
obj.speak();      // speak
```

- Java is a strongly typed language
 - Reference variable obj, is type **Animal**
- Therefore, variable obj:
 - uses ONLY **Animal** API
 - Calls *speak()* from superclass API **Animal**
 - Executes *speak()* from derived object **Dog**
 - **Dog** *speak()* implementation overrides **Animal** *speak()* implementation via polymorphism

Using Polymorphism

- Use of an **Animal** container is significant
 - Use **Animal** class as an *abstracting API*
 - Uses ALL objects as if an **Animal** object.
 - Using **API** provides *Functionality Hiding*

Using Polymorphism

- Using API provides *Functionality Hiding*
 - API to Specify **What** without specifying **How**
 - Use an Abstract class or Interface to specify an API for derived classes which **MUST** implement missing functionality
 - API to Specify **What** with specifying **How**
 - Concrete **Animal** class allows API to provide a DEFAULT implementation (**How**) IF derived class does not provide it's own implementation (**override**)

Using Polymorphism

- Using API provides *Functionality Hiding*
 - Hides specific implementation of derived classes implementing **Animal API**
- Using Animal API also provides **Loose Coupling**
 - Derived class details do not have to be used to Speak()

| Loose Coupling of classes:
classes are independent of each other
classes are used outside only with common interface
classes does not know each other's implementation

Using Polymorphism

- Use of an **Animal** container is significant
 - Allows Driver to be designed to use Animal objects **ONLY** and only by Animal container **FUNCTIONALITY HIDING**
 - Driver design is independent of any derived child classes of Animal (but Driver can use them all because of ‘*IS-A*’ relationship)

Using Polymorphism

- Use of an **Animal** container is significant
 - Driver is compatible with any future design extensions as long as they are derived from the Animal class
 - Implementing and overriding info method provides the new functionality

Using Polymorphism

- Java keyword ‘**final**’
 - Java allows a subclass to **override** any superclass method *by default*.
 - The subclass must provide it’s own implementation in one of it’s methods that has an **IDENTICAL** signature to the superclass method to be overriden
 - Java allows a superclass to disable this feature and not allow a subclass to override it’s method
 - The superclass must use the keyword ‘**final**’ to disable function overriding

CSYE 6200

Concepts of Object Oriented Design

Explosion Class

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Inheritance
 - 2. Java Abstract Class
 - 3. Java Interface

Relationships

- Association
 - Aggregation
 - “Has-A”
- Generalization
 - Inheritance
 - “Is-A”

Relationships

- Association

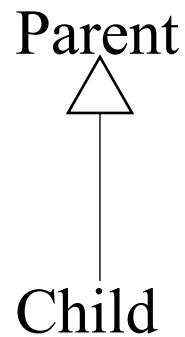
Teacher $\xleftarrow{1} \xrightarrow{1}$ Classroom

Student $\xleftarrow{10..*} \xrightarrow{1}$ School

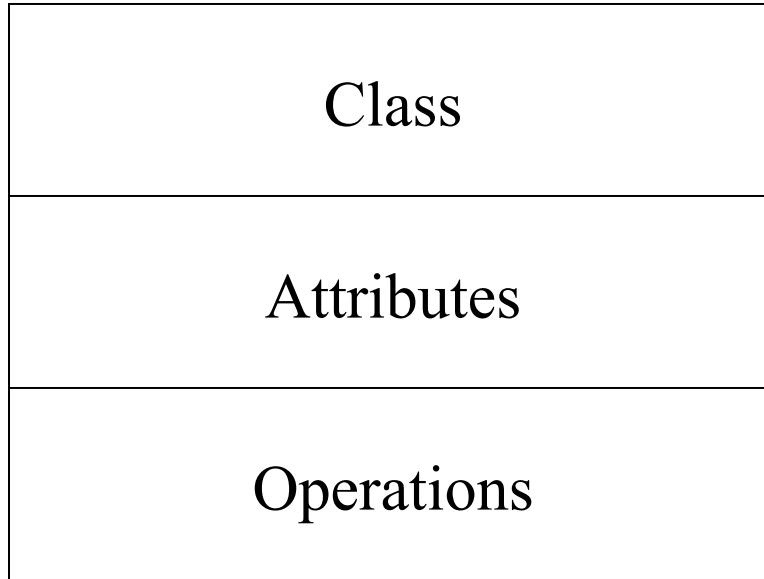
Employee $\xleftarrow{1..*} \xrightarrow{1}$ Corporation

Relationships

- Generalization

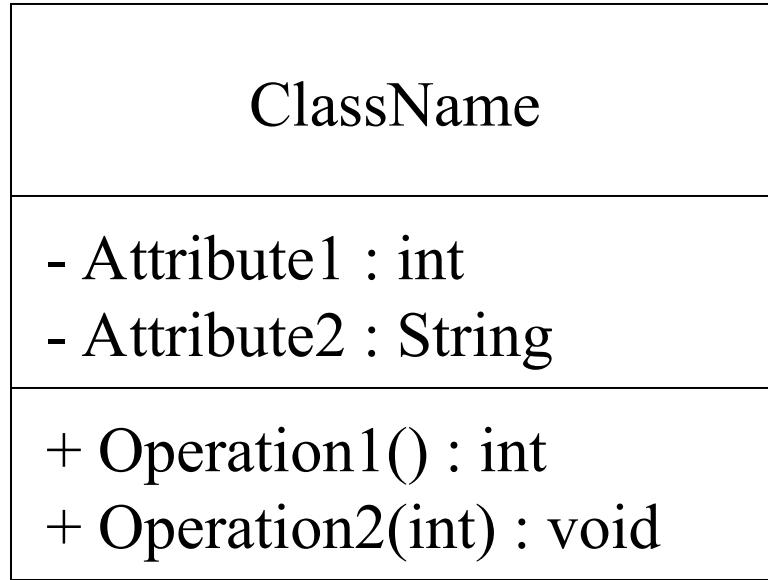


UML Class Diagram

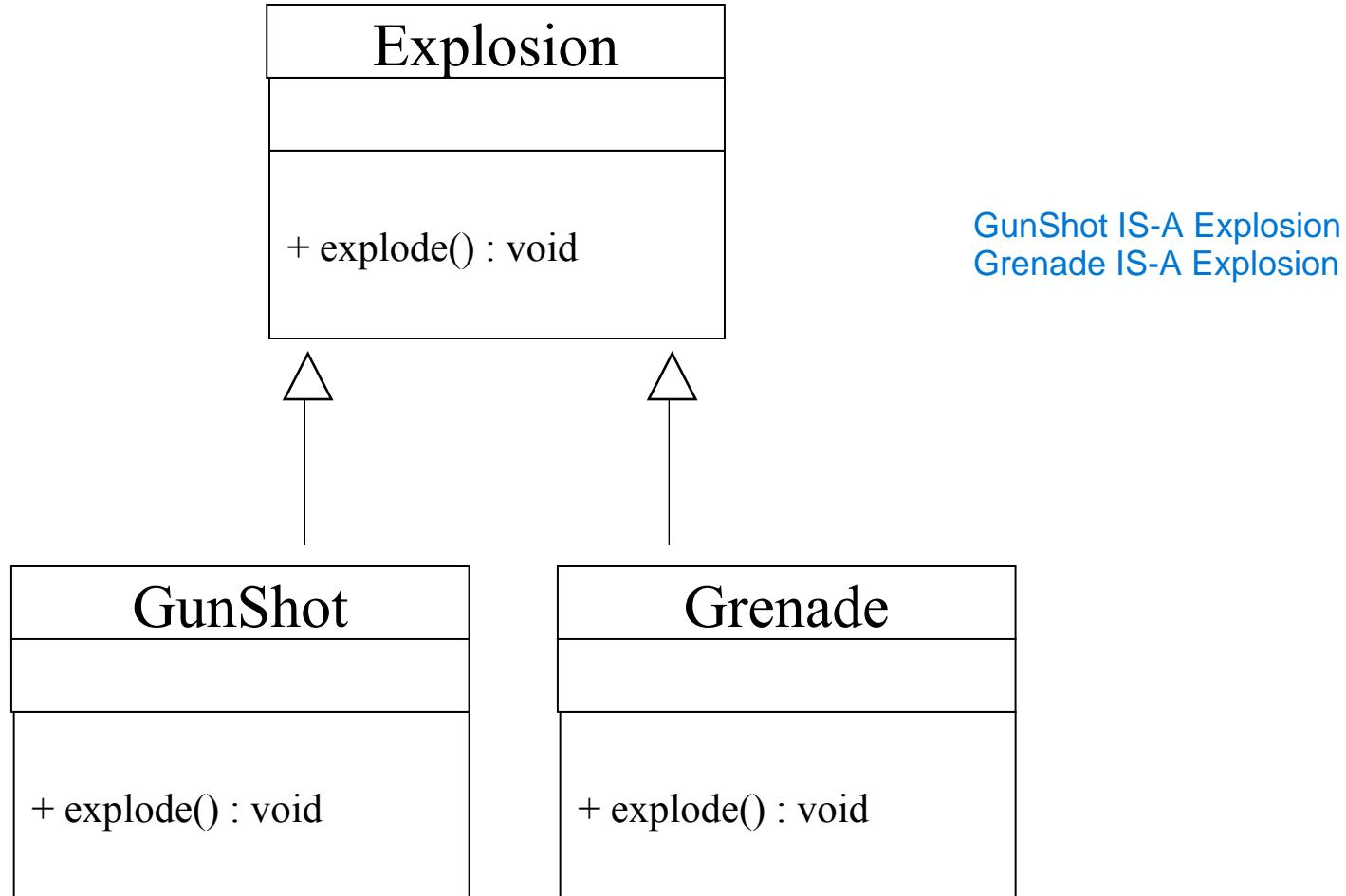


UML Class Diagram

- private
+ public



Explosion Class Diagram



Explosion Class

```
public class Explosion {  
    public void explode() {  
        System.out.println(  
            "Explosion [** EXPLODE **] !!!"  
        );  
    }  
}
```

GunShot Class

```
public class GunShot extends Explosion {  
    @Override  
    public void explode() {  
        System.out.println(  
            "GunShot [** BANG **] !!!"  
        );  
    }  
}
```

GunShot Class

```
public class GunShot extends Explosion {  
}
```

- If **GunShot** class does not provide it's own *explode()* implementation
 - inherits default *explode()* implementation from **Explosion**

GunShot Class

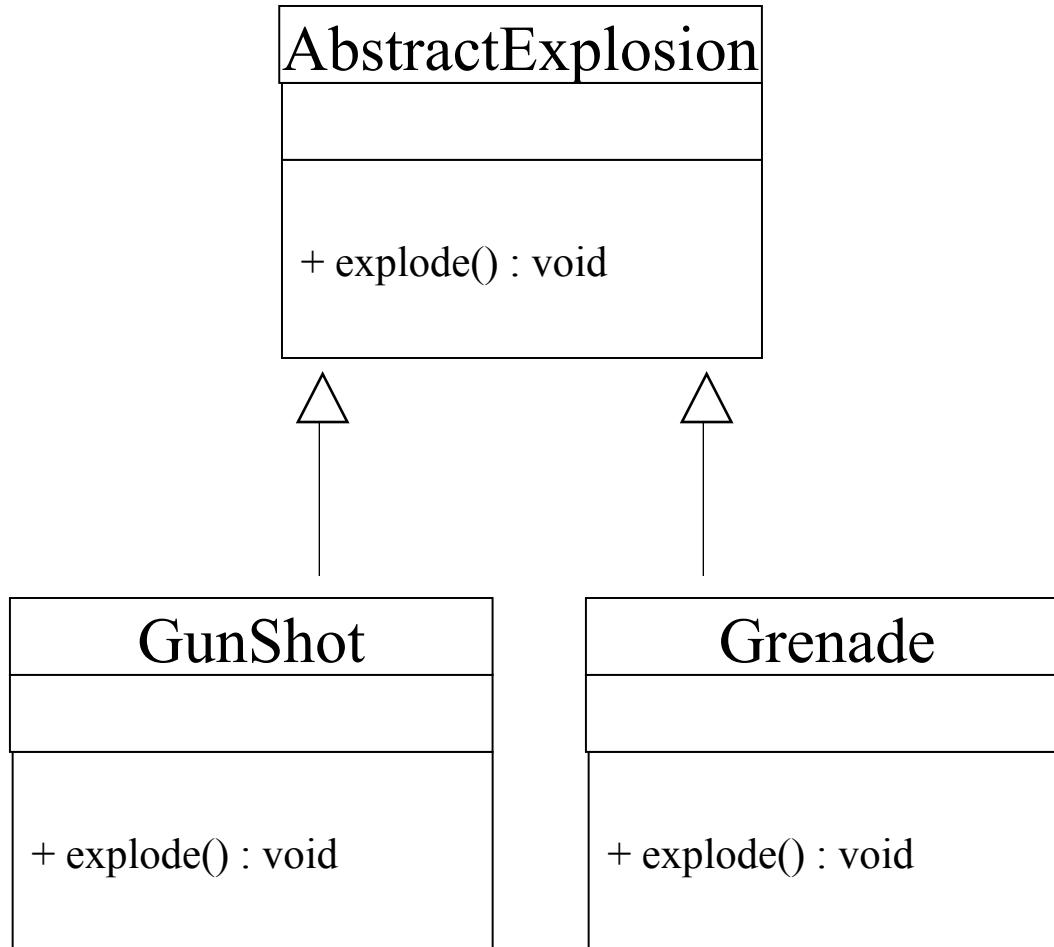
```
public class GunShot extends Explosion {  
    public void explode() {  
        System.out.println(  
            "Explosion [** EXPLODE **] !!!"  
        );  
    }  
}
```

- As if GunShot were written as above
 - **GunShot** can inherit **Explosion** *explode()*
-

Grenade Class

```
public class Grenade extends Explosion {  
    @Override  
    public void explode() {  
        System.out.println(  
            "Grenade [** SPLATTER **] !!!"  
        );  
    }  
}
```

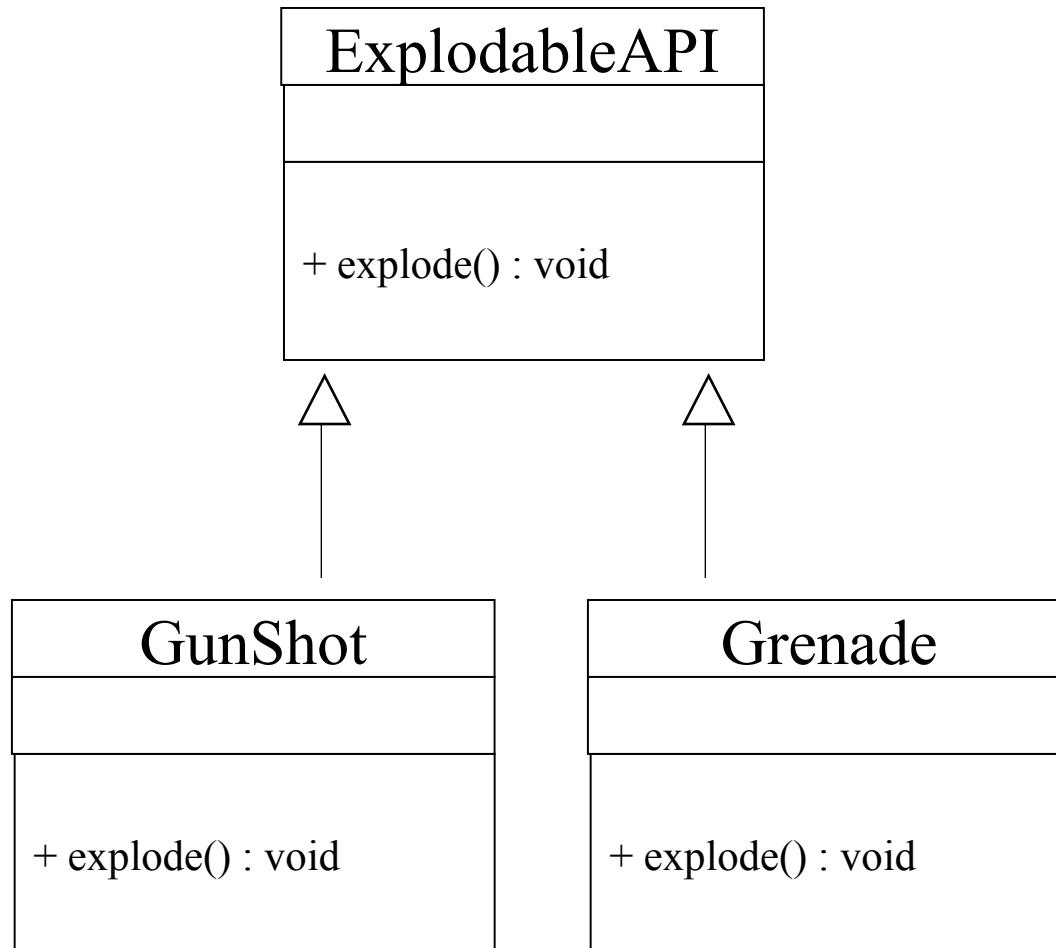
AbstractExplosion Class Diagram



AbstractExplosion Class

```
public abstract class AbstractExplosion {  
    public abstract void explode();  
}
```

ExplodableAPI Class Diagram



ExplodableAPI Interface

```
public interface ExplodableAPI {  
    void explode();  
}
```

Class

- Class
 - class MyName [extends MySuperClass]
[implements MyInterface] {
 - Data
 - Constructor
 - Method
 - }

Concrete Class

- Class
 - Concrete
 - Declared: public, private or protected members
 - Fully implemented methods
 - Data
 - final or non final
 - static or non-static
 - Can be instantiated to create objects

Abstract Class

- Class
 - Abstract
 - Declared: public, private or protected members
 - Partially implemented
 - Contains one or more abstract methods
 - Data
 - final or non final
 - static or non-static
 - Must be extended (keyword extends)
 - Cannot be instantiated (without completing implementation)

Interface

- Class
 - Interface
 - Contains ONLY public methods
 - ‘abstract’ methods are unimplemented
 - Java 8 ‘default’ Methods are implemented
 - Java 8 ‘static’ Methods are implemented
 - Data
 - static (class variables) ONLY
 - final (immutable constant values) ONLY
 - Must be implemented (keyword implements)
 - Cannot be instantiated (without completing implementation)

Class (cont'd)

- Class
 - Data
 - Attribute
 - Field
 - Constructor
 - One or more special Method to instantiate objects
 - Method
 - Function
 - Operation
 - Behavior

Class (cont'd)

- Class
 - Data declaration
 - [static] [final] [public | protected | private] type name
 - [= initializer] ;

Class (cont'd)

- Class
 - Class Constructor declaration
 - [static] [final] [public | protected | private]
 - ClassName()
 - { constructor method body } ;

Class (cont'd)

- Class
 - Method declaration
 - [static] [final] [public | protected | private]
 - [returnType | void] methodName()
 - { method body } ;

Java Interface

- Java Interface
 - Reference type, similar to a class
 - Contains ONLY:
 - Data
 - Public Static Final (Constants)
 - Methods:
 - ALL METHODS IMPLICITLY PUBLIC
 - » Keyword public may be omitted from methods.
 - abstract methods (no body)
 - Static methods
 - Default methods (Java 8)

Java Interface

- Java Interface
 - Public:
 - Public interface is accessible to all classes
 - Otherwise, only accessible to classes in same package
 - Keyword interface
 - Must be implemented by a class
 - Class MUST implement all interface methods
 - Can be extended by other interfaces

Java Interface

- Java Interface Use
 - Must be implemented by a class
 - Class MUST implement all interface methods
 - Can be extended by another interface

Java Interface

- Java Interface as API
- “The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. **APIs are also common in commercial software products.** Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's **API is made public (to its customers), its implementation of the API is kept as a closely guarded secret**—in fact, it **may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.**”

<https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>

Java Interface

- Java Interface Examples
 - Comparable
 - Implement to make your class sortable by default in natural order.
 - Comparator
 - Implement and use to specify a specific sort order.
 - Runnable
 - Implement to make your class executable on a new thread.
 - ExplodableAPI

CSYE 6200

Concepts of Object Oriented Design

Generics

Daniel Peters

d.peters@neu.edu

Generics

- Lecture
 1. List<Object>
 2. List<>
 3. List<T>
 4. List<?>
 5. List<? extends Person>

Parameterized Types

- Collections of elements of a specific type
 - Specify type of element in parameterized type

```
List<Student> students = new  
Vector<Student>();
```

- Type Inference:
 - Java SE 7 Compiler can infer or determine type:

```
List<Student> students = new Vector<>();
```

-
- “In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that $G<Foo>$ is a subtype of $G<Bar>$. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.”
 - <https://docs.oracle.com/javase/tutorial/extras/generics/subtype.html>
-

- In general, if *Child* is a subtype (subclass or subinterface) of *Parent*, and G is some generic type declaration, it is **not** the case that $G<Child>$ is a subtype of $G<Parent>$.
This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.
- <https://docs.oracle.com/javase/tutorial/extras/generics/subtype.html>
- Danny paraphrase

-
1. If Class ***Child*** is a subtype (subclass or derived from) Class **Parent**,
 2. AND G is some generic type (example **List<G>** or **Vector<G>**),
 3. **List< Child >** is **NOT** a subtype of **List< Parent >**.

Generic Type

- Class name<T1, T2, ... Tn> {/* ... */}
- Pithy (single char) naming convention
 - Formal Type
 - Single, **uppercase** character naming convention
 - E
 - K – Key
 - N – Number
 - T – Type
 - V – Value
 - S,U,V etc. 2nd, 3rd, 4th types

Generic Type Declaration

- Generic Class Declaration
 - Use T as a generic type parameter
 - It is used as a place holder to be replaced by Actual Specific Type when Generic Class is used
 - NOTE: Prior to JDK 5.0 legacy code used raw type UNTIL classes became generic, i.e. a concrete type (e.g. Integer) can be applied

```
class Box<T> {  
    ...  
}
```

Generic Type Invocation

- Generic Class Invocation:
 - Replace formal type T by substituting *Integer* as generic type argument as a **parameterized type**
 - Substitute any **non-primitive** type
 - **Reference Types ONLY**

```
class Box<Integer> // Integer class wrapper  
class Box<Double> // Double class wrapper  
NOT class Box<double> // primitive double type
```

Generic Type Benefits

- Type Abstraction
- Promote Code Reuse
 - Test and Development efficiency
 - Consistency
 - Flexibility

```
List<Integer> list1 = new ArrayList<>();  
List<?> list2 = new ArrayList<>();
```

```
list1.add(5); // This is valid because list1 is specifically a List of Integers.  
list2.add(5); // This is not valid because list2 is a wildcard type, and we don't know its exact type.
```

```
Integer item1 = list1.get(0); // Valid, as we know list1 is a List of Integers.  
Object item2 = list2.get(0); // Valid, but the type is Object because we used a wildcard.
```

Wildcard

“In generic code, the question mark (?), called the *wildcard*, represents an unknown type.”

<https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html>

Foo<T> is a generic type where T is a type parameter that can be specified when you create an instance of the Foo class. It represents a specific, known type.

When you use Foo<T>, you are indicating that you want to use Foo with a specific type T. For example, Foo<Integer> would be a Foo that works with Integer objects.

Foo<?> is a wildcard type in Java. It's used when you want to work with an unknown type or when you don't care about the specific type argument. It's often used in situations where you need to deal with generic types, but you want to provide more flexibility in the type of elements you can work with.

When you use Foo<?>, you are saying that you are willing to accept any type argument for Foo. This is similar to saying "I don't care about the exact type; it can be anything."

Wildcard

“To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its *upper bound*.”

<https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html>

Wildcard: List<? extends Foo>

- Upper bounded Type

<? extends Foo>

- Foo is any type
- Matches Foo
 - (and classes derived from Foo)
 - IS-A relationship
- Matches **ANY SUBTYPE** of Foo.

Wildcard: List<? extends Explosion>

- Upper bounded Type

<? **extends** Explosion>

- Explosion is a explosion API concrete class
- Matches Explosion super class
- Matches **ANY SUBTYPE** of Explosion.
 - GunShot Is-A Explosion
 - Grenade Is-A Explosion

Wildcard: List<? extends Number>

- Example: **sumOfList**

```
public static double sumOfList(List<? extends  
    Number> list) {  
    double sum = 0.0;  
    for (Number n : list)  
        sum += n.doubleValue();  
    return sum;  
}
```

Wildcard: List<? extends Number>

- Example: **sumOfList**

```
public static double sumOfList(List<?  
    extends Number> list) {
```

- Matches a list of any **subtype** of Number (Integer, Short, Double, Long, etc.).

Wildcard: List<? extends Number>

- Example: **sumOfList**

```
List<Integer> ints = Arrays.asList(1, 2, 3);
System.out.println("sum = "
    + sumOfList(ints));
```

- List of Integer objects:
1, 2, 3
- Prints sum = 6.0:

Wildcard: List<? extends Number>

- Example: **sumOfList**

```
List<Double> d = Arrays.asList(1.2, 2.3, 3.5);  
System.out.println("sum = " +  
    sumOfList(d));
```

- List of Double objects:
 1.2, 2.3, 3.5
- Prints sum = 7.0:

Wildcard: List<?>

- Unbounded Type: List<?>
- “Because for any concrete type A, List<A> is a subtype of List<?> ...”
- Member of every type: null

Wildcard: List<?>

- Note List<Object> and List<?> are NOT THE SAME.
 - List<Object>
 - Can insert Object
 - Can insert (add to List) ANY subtype of Object
 - List<?>
 - Can ONLY insert (add to List) null
- Unbounded wildcard ? useful when NOT inserting (add to List)

Wildcard: List<?>

- Example: printList

```
public static void printList(List<?> list) {  
    for (Object elem: list)  
        System.out.print(elem + " ");  
    System.out.println();  
} // end printList
```

- Leverage Object.toString()

Wildcard: List<?>

- Unknown (Unbounded) Type
- Member of every type: null
- Useful:
 - If you are writing a method that can be implemented using functionality provided in the **Object** class.
 - When the code is using methods in the generic class that don't depend on the type parameter.
 - List.size
 - List.clear.

Wildcard: List<? super Foo>

- Lower bounded Type

<? **super** Foo>

- Foo is any type
- Matches Foo
- Matches **ANY SUPER TYPE** of Foo.

- Name **TWO** classes? Foo and Object
-

Wildcard: List<? super Foo>

- Example: **addNumbers**

```
public static void addNumbers(List<? super
Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    } // end for
} // end addNumbers
```

Wildcard: List<? super Foo>

- Example: **addNumbers**

```
public static void addNumbers(List<? super
    Integer> list) {
    ...
}
```

// end addNumbers

- Matches a list of any **supertype** of Integer (Integer, Number and Object).
-

Generic Class Example

```
public class ShowState<E> {  
    private List<E> objects = new ArrayList<E>();  
    public void addObject(E obj) {  
        this.objects.add(obj);  
    }  
    public void showObjects(String title) {  
        System.out.println(title);  
        this.objects.stream()  
            .forEach(System.out::println); // call toString()  
        System.out.println(objects.size() + " objects.");  
    }  
} // Why can you System.out::println an object from ANY class? Ans. toString()
```

Generic Class Example

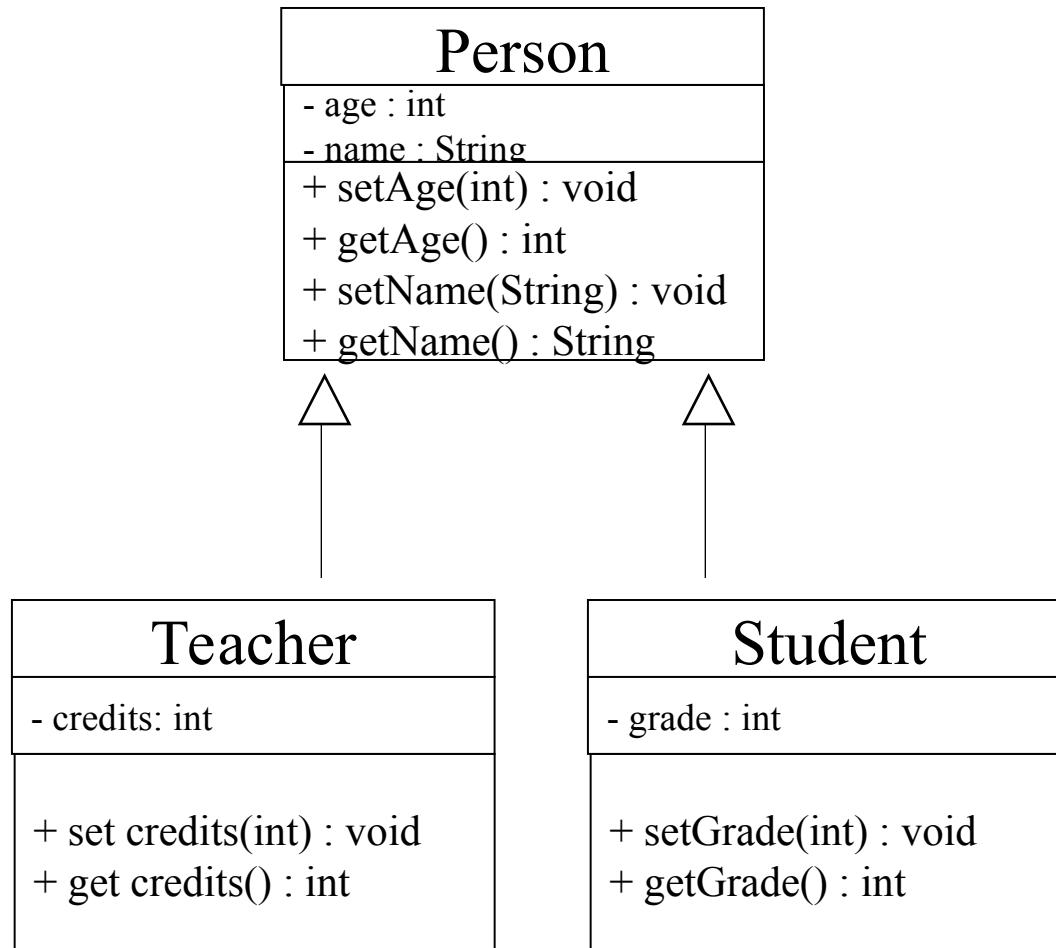
```
public class ShowState<E> {  
    private List<E> objects = new ArrayList<E>();  
    public void addObject(E obj) {  
        this.objects.add(obj);  
    }  
    public void showObjects(String title) {  
        System.out.println(title);  
        this.objects.stream()  
            .forEach(o -> System.out.println(o.toString()));  
        System.out.println(objects.size() + " objects.");  
    }  
}
```

} What can implement Consumer Interface? Lambda, anon., concrete classes

Java type casting

- instanceof
- JavaClass class.cast
- (type)

Class Diagram



Type casting example

```
Student s = new Student(); // Student object  
Student s2 = null; // TYPE: ref to student obj  
Person p = s; // Student IS-A Person  
// does p reference a Student object?
```

Upcasting
Typecasting child type to parent type
implicit

```
if (p instanceof Student) {  
    s2 = (Student) p; // if true, downcast  
}
```

Downcasting
typecasting parent type to child type
not implicit

- Example of downcasting (bad) with casting

Type casting example

```
Student s = new Student();
```

```
Student s2 = null;
```

```
Person p = s;      // ok, Student IS-A Person
```

```
if (p instanceof Student) {
```

```
    s2 = p; // true, but illegal without cast
```

```
}
```

- Inheritance IS-A relationship

- Child IS-A Parent

- Parent is never a Child

Type casting example

```
SwingControlDemo sw3 = new  
SwingControlDemo();
```

```
Person p = null;
```

```
// can't cast just any object
```

```
p = (Student) sw3; // NOT a Student object
```

GUI Updater Example

```
package edu.neu.csye6200.util;  
import java.util.ArrayList;  
import javax.swing.*;  
import javax.swing.JComboBox;  
  
public class TextAreaUpdaterTask<T extends JComponent>  
implements Runnable {  
  
    /**  
     * construct a Runnable updater for the text of one GUI component  
     * @param guiComponent targeted GUI component for text update  
     */  
  
    ...  
}  


---


```

GUI Updater Example

```
public TextAreaUpdaterTask(T guiComponent) {  
    this.guiComponent = guiComponent;  
}  
  
JButton b;  
JComboBox c;  
JLabel l;  
JTextArea t;  
JTextField f;  
  
String message; // message to update gui component  
ArrayList list; // list of messages to update gui component  
T guiComponent; // gui components to update with message
```

GUI Updater Example

```
/**  
 * provide text for updating GUI component  
 *  
 * @param message text for GUI component update  
 */  
  
public void setMessage(String message) {  
    this.message = message;  
}
```

GUI Updater Example

```
/**  
 *  
 * @param aList list of messages to update GUI component  
 */  
  
public void setMessage(ArrayList aList) {  
    this.list = aList;  
}  
  
private <S> S convert(Class<S> cls, Object o) {  
    return cls.cast(o);  
}
```

GUI Updater Example

```
@Override  
public void run() {  
    if (guiComponent instanceof JButton) {  
        b = (JButton) guiComponent;  
        b.setText(message); // update button text  
    }  
  
    if (guiComponent instanceof JComboBox) {  
        c = convert(JComboBox.class, guiComponent);  
        c.removeAllItems();  
        for (Object item : list) {  
            c.addItem(item);  
        }  
    }  
}
```

GUI Updater Example

```
if (guiComponent instanceof JLabel) {  
    l = (JLabel) guiComponent;  
    l.setText(message); // update label text  
}  
  
if (guiComponent instanceof JTextArea) {  
    t = (JTextArea) guiComponent;  
    t.setText(message); // update text area contents  
}  
  
if (guiComponent instanceof JTextField) {  
    f = (JTextField) guiComponent;  
    f.setText(message); // update text field  
}  
}
```

Online Links

- Oracle:
 - Generics:
<https://docs.oracle.com/javase/tutorial/java/generics/index.html>
 - Guidelines for Wildcard Use:
<https://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html>
 - Restrictions on Generics:
<https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>

CSYE 6200

Concepts of Object Oriented Design

Java Collections: Containers and Algorithms

Daniel Peters

d.peters@neu.edu

- Lecture

- 1. List

- Vector, ArrayList, LinkedList

- 2. Map

- HashMap, HashTable, TreeMap

- 3. Set

- HashSet, TreeSet, LinkedHashSet

- 4. Algorithms

Java Collections

- Java Collections Framework
 - Collections (a.k.a. Containers) contains a group of objects (i.e. one or more elements or items) of the same type
 - Primitive types are NOT ALLOWED
 - Requires boxing (Auto-Boxing):
 - Wrap ‘int’ in Integer class
 - Wrap ‘double’ in Double class
- Unboxing conversion that the Java compiler makes from its object of wrapper type to its corresponding primitive type
- ```
ex: Integer a = 2;
int[] arrayA = new int[1];
arrayA[0] = a; // unboxing
```
- Automatic conversion that the Java compiler makes from its primitive type to corresponding wrapper class type
- ```
ex: int a = 1;  
List<Integer> listA = new ArrayList<>();  
listA.add(a); // autoboxing
```

Java Collections

- Java Collections Framework
 - Online Information
 - <https://docs.oracle.com/javase/tutorial/collections/>
 - <https://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html>
 - <http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>
 - <http://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html>

Java Collection Terms

- Unmodifiable
 - Do not support add, remove, clear methods
- Immutable
 - Guarantee no change to item will be visible
- Fixed-size
 - Guarantee size remains constant
- Random-access
 - Same time to access to any item
 - Opposite of Sequential access

Java List

- List
 - Interface: **ordered** collection (i.e. sequence)
 - Implemented by several collection classes
 - Best Practice: Use List API
 - names.add("James");
 - facilitates change to collection class if required
(ArrayList => Vector => LinkedList)
 - Declare variables as List; instantiate Object of choice (ArrayList, Vector, LinkedList, etc.)
 - List<String> names = new ArrayList<String>();
 - List<String> names = new Vector<String>();

Java Collection classes

- Vector
 - Synchronized (thread safe)
 - Resizable random-access List implementation
 - Better than fixed-size array
- ArrayList
 - Unsynchronized Vector (essentially)
 - NOT Thread safe
 - Executes Faster than Vector

Java LinkedList class

- **LinkedList**
 - Unsynchronized
 - Faster than ArrayList
 - resizable List implementation
 - Queue implementation
 - LIFO
 - Last In First Out (i.e. for stack use)
 - Dequeue implementation
 - FIFO
 - First In First Out

Vector Container

- Implements the List interface
- Can be instantiated
- Can grow and shrink as needed
 - Growable array
 - Storage size increases in capacityIncrement chunks
- Ordered collection (i.e. sequence)
- Supports random access like an array
- Like ArrayList but is thread safe
(synchronized)
- Slower than ArrayList

ArrayList Container

- Java Collections Framework
- Implements the List interface
- Can be instantiated
- Can grow and shrink as needed
- Ordered collection (i.e. sequence)
- Supports random access
- Like Vector but not thread safe
(unsynchronized)
- Faster than Vector

LinkedList Container

- Implements the List interface
- Can be instantiated
- Can grow and shrink as needed
- Ordered collection (i.e. sequence)
- Does NOT support random access
- NOT thread safe (unsynchronized)
 - `List list = Collections.synchronizedList(new
LinkedList(...));`
- Double Linked List implementation

Java Collection classes

- Queue
 - Interface: LIFO: add(), remove(), peek(), poll()
- Dequeue
 - Interface: FIFO
 - Double-ended Queue; extends Queue

Java Collection classes

- Map
 - Interface for **Associative** containers
 - Contains Key Value (K,V) pairs
 - Unique key mapped to each item

Java Collection classes

- **HashMap**
 - Implements Map interface
 - Unsynchronized: NOT Thread-safe
 - Allows NULL as key or value
 - Unordered
- **HashTable**
 - Implements Map interface
 - Synchronized: Thread-safe
 - Slower than HashMap

Java Collection classes

- TreeMap
 - Implements Map interface
 - Unsynchronized: NOT Thread-safe
 - `SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));`
 - Ordered (natural order or Comparator)

HashMap Container

- Java Collections Framework
- Implements the Map interface
- Can be instantiated
- Can grow and shrink as needed
- Unordered Associative collection (i.e. Key, Value pairs)
- Accepts null (key or value)
- NOT thread safe (unsynchronized)
 - `Map m = Collections.synchronizedMap(new HashMap(...));`

HashMap Container

```
Map<Integer, String>m = new HashMap<>();  
m.put(1,"one");           // auto-boxing int into Integer object  
m.put(3,"three");  
m.put(5,"five");  
m.put(null, "nullKey");    // hashMap accepts null key  
m.put(7, null); // HashMap accepts null value  
System.out.println("5 is the integer index for the string " +  
m.get(5));  
System.out.println("Display map keys: ");  
for (Integer key : m.keySet()) {  
    System.out.print(key + ", ");  
}  
}
```

HashMap Container

```
Map<Integer, String>m = new HashMap<>();  
...  
for (Integer key : m.keySet()) {  
    System.out.print(key + ", ");  
}  
System.out.println("Display map values: ");  
for (String value : m.values()) {  
    System.out.print(value + ", ");  
}  
System.out.println(m.size() + " Elements in map: " + m);
```

HashTable Container

- Java Collections Framework
- Implements the Map interface
- Can be instantiated
- Can grow and shrink as needed
- Unordered Associative collection (i.e. Key, Value pairs)
- DOES NOT accept null (key or value)
- Synchronized: thread safe
 - Highly Concurrent, use ConcurrentHashMap

TreeMap Container

- Java Collections Framework
- Implements the Map interface
- Can be instantiated
- Can grow and shrink as needed
- Ordered Associative collection (i.e. Key, Value pairs)
- DOES NOT accept null (key or value)
- NOT thread safe (unsynchronized)
 - `SortedMap m =
Collections.synchronizedSortedMap(new
TreeMap(...));`

TreeMap Example

```
Map<Integer, String> m =  
    new TreeMap<Integer, String>();
```

```
m.put(19, "Dan");  
m.put(21, "James");  
m.put(3, "Baby Sue");  
System.out.println("3 is the correct age for " +  
    m.get(3));
```

TreeMap Example

```
Map<Integer, String> m =  
    new TreeMap<Integer, String>();  
  
m.put(19, "Dan");  
m.put(21, "James");  
m.put(3, "Baby Sue");  
System.out.println("3 is the correct age for " +  
    m.get(3));
```

Java Set

- Set – NO Duplicate elements
 - Interface: unordered collection (i.e. sequence)
 - Implemented by several collection classes
 - Best Practice: Use Set API
 - names.add("James");
 - facilitates change to collection class if required (HashSet => TreeSet => LinkedHashSet)
 - Declare variables as Set; instantiate Object of choice (HashSet, TreeSet, LinkedHashSet)
 - Set<String> names = new HashSet<String>();
 - Set<String> names = new TreeSet<String>();
 - Set<String> names = new LinkedHashSet<String>();

Java Set

- Set

<https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html>

Java Collection classes

- HashSet
 - Unsynchronized (NOT thread safe)
 - Unordered
 - Faster than Ordered TreeSet
 - Backed by HashMap implementation of HashTable
 - No Duplicate entries
 - Null elements allowed

Java Collection classes

- TreeSet
 - Unsynchronized (NOT thread safe)
 - Ordered (natural order)
 - Slower than unordered HashSet
 - Slower than ordered LinkedHashSet
 - Backed by TreeMap
 - No Duplicate entries
 - Null entries NOT allowed (exception thrown)

Java Collection classes

- **LinkedHashSet**
 - Unsynchronized (NOT thread safe)
 - Double-Linked List implementation
 - “Iteration over a LinkedHashSet requires time proportional to the *size* of the set,”
 - Insertion Order (order inserted)
 - Faster than TreeSet
 - No Duplicate entries
 - Null elements allowed

Arrays.asList

- asList: array to list
 - Random Access list
 - Implements RandomAccess interface
 - Returns a Fixed List
 - Cannot grow or shrink
 - List is Backed by Array
 - Writes to list also writes to Array
 - List View of original array

Arrays.asList

```
String[] names = { "Jen", "Zac", "Dan" };  
List<String> listOriginal = Arrays.asList(names);
```

- `listOriginal.add("Mary");`
 - Run-Time EXCEPTION:
 - cannot add to fixed size list

Arrays.asList

```
String[] names = { "Jen", "Zac", "Dan" };
```

```
List<String> listOriginal = Arrays.asList(names);
```

```
listOriginal.set(2, "Daniel");// writes through to array
```

- Both array names and List list contain identical data:
 - Jen, Zac, Daniel

Contains: To Search List

- `java.util.List`
- `contains`
 - Returns *true* if this list contains the specified element.

```
if ( list.contains(target) ) {  
    // target found in list  
    index = list.indexOf(target);  
}
```

Find Dan

```
String[] names = { "Sam", "Dan", "Jim" };
List<String> list = new ArrayList<>(Arrays.asList(names));

if (list.contains("Dan")) {
    System.out.print("FOUND DAN! ");
    index = list.indexOf(target);
    System.out.println("# " + ++index + " in
the list");
}
```

Console Output

FOUND DAN! # 2 in the list

Find Guitar Item

```
List<Item> list = new ArrayList<Item>();  
Item car = new Item(2, "Car");  
Item home = new Item(1, "Home");  
Item guitar = new Item(3, "Guitar");  
list.add(car);  
list.add(home);  
list.add(guitar);  
if (list.contains(guitar)) {  
    index = list.indexOf(target);  
    System.out.println("FOUND GUITAR! # " + ++index + " in  
the list");    Found Guitar! #2 in the list  
}
```

Collection Comparision

| Collection | Interface | Thread Safe | Resizable | Access | Quickness | Ordering | Duplicates | Null |
|-------------------|---------------|---------------------|----------------|--------|-----------------------|-----------|---------------|-------------|
| ArrayList | List | Unsynchronized | Not Resizable | O(1) | Faster than Vector | Ordered | Duplicates | |
| Vector | List | Synchronized | Resizable | O(1) | — | Ordered | Duplicates | Not allowed |
| LinkedList | List | Unsynchronized | Resizable | O(n) | Faster than ArrayList | Ordered | Duplicates | Allowed |
| DoubleLinkedList | LinkedList | — | Resizable | O(n) | — | — | — | — |
| Queue | LIFO | — | Ends Resizable | O(n) | — | Ordered | Duplicates | Not allowed |
| Dequeue | FIFO | — | Ends Resizable | O(n) | — | Ordered | Duplicates | Not allowed |
| HashMap | Map | Unsynchronized | Resizable | O(1) | Faster than HashTable | Unordered | — | Allowed |
| HashTable | Map | Synchronized | Resizable | O(1) | — | Unordered | — | Not allowed |
| ConcurrentHashMap | Map | Highly Synchronized | Resizable | — | — | — | — | — |
| TreeMap | Map | Unsynchronized | Resizable | — | — | Ordered | — | Not allowed |
| HashSet | Set, HashMap | Unsynchronized | Resizable | O(1) | Faster than TreeSet | Unordered | No duplicates | Allowed |
| TreeSet | Set, TreeMap | Unsynchronized | Resizable | O(1) | — | Ordered | No duplicates | Not Allowed |
| LinkedHashSet | Set, DoubleLL | Unsynchronized | Resizable | — | Faster than TreeSet | — | — | Allowed |

Array [] has a smaller memory footprint than ArrayList

FindItemInList

```
public Item findItemInList(List<Item> list, Item target) {  
    Item foundItem = null;  
    Iterator<Item> it = list.iterator();  
    while (it.hasNext()) {  
        foundItem = it.next(); // get next Item in list  
        if (list.contains(target)) {  
            break; // found target Item in list  
        } else {  
            foundItem = null;  
        }  
    }  
    return foundItem; // return Item found in list  
}
```

Algorithms

- `java.util`
- `Arrays` class
 - `asList`
 - `Sort`
 - `Comparable`
 - `Comparator`

Sort Comparable Example

```
Integer[] numbers = { 9, 7, 23, 3, 5 };  
java.util.Arrays.sort(numbers);
```

- Sorted array contains:
 - 3, 5, 7, 9, 23
 - Comparable interface provides class with support for sort() natural order
 - Integer class implements natural order (via Comparable interface)
-

Sort Comparable Example

```
String[] names = { "sam", "dan", "jim" };
```

```
List<String> list = new
```

```
ArrayList<>(Arrays.asList(names));
```

```
java.util.Collections.sort(list);
```

- Sorted List contains:

- dan, jim, sam

- String implements natural order (Comparable interface)

Sort Algorithm

```
List<String> list =  
new ArrayList<String>();
```

... .

```
list.sort(null);
```

- Sorted List contains:
 - dan, jim, sam
- String implements Comparable for natural order sort (comparator not required).

Algorithms

- `java.util`
- Collections class
 - Contains
 - Sort
 - Comparator
- class Student
 - Attributes: Id, name, GPA
 - Sort by class attributes

Comparator

- Package: java.util
- Interface
 - java class ‘implements’ Comparator

Comparator class

```
class AlphabetizeStrings implements  
Comparator<String> {  
  
    @Override  
    public int compare(String o1, String o2)  
    { // sort in Natural order  
        return o1.compareTo(o2);  
    }  
}
```

Comparator Example

```
String[] names = { "Jen", "Zac", "Dan" };
```

```
Arrays.sort(names, new AlphabetizeStrings());
```

- *Array names contains:*
 - Dan, Jen, Zac
- Comparator directs sort to “Sort-this-way”.
- Class AlphabetizeStrings implements Comparator for “Sort-this-way” for Strings.

Anonymous Comparator Example 1

```
String[] names = { "Jen", "Zac", "Dan" };
Arrays.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);      // lexicographic order
    }
});    // Dan, Jen, Zac
```

- *Array names contains:*
 - Dan, Jen, Zac
-

Anonymous Comparator Example 2

```
List<String> names = new ArrayList<String>();
```

```
// add elements to collection
names.add("Adam");
names.add("Eve");
names.add("Marilyn");
names.add("Robin");
names.add("William");
names.add("Devon");
names.add("Sylvester");
```

Anonymous Comparator Example 2

```
List<String> names =  
new ArrayList<String>();  
  
...  
Collections.sort(names, new  
Comparator<String>() {  
    // sort collection by element length  
    public int compare(String o1, String o2) {  
        return o1.compareTo(o2);  
    }  
});
```

Sort Student Example

```
List<Student> students =  
new ArrayList<Student>();
```

```
Collections.sort(students);
```

- Student **MUST** implement Comparable for natural order sort (comparator not required)

Sort Student Example

```
public class Student extends Person implements  
Comparable<Student> {  
    ...  
    // Implement compareTo for Comparable  
    @Override public int compareTo(Student obj) {  
        // Student natural order: sort by GPA  
        return Double.compare(this.getGpa(),  
                           obj.getGpa());  
    }  
}
```

Sort Student Example

```
List<Student> students =  
new ArrayList<Student>();
```

```
Collections.sort(students, Comparator);
```

- Specific Comparator supplied to direct sort (Comparable implementation not required).
- Use Comparators to provide additional specific sort orders for your object.

Sort Student Example

```
List<Student> students =  
new ArrayList<Student>();
```

```
students.sort(Comparator);
```

- Specific Comparator supplied to direct sort (Comparable implementation not required).
- Use Comparators to provide additional specific sort orders for your object.

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    /**  
     * Implement Comparable so Student can be sorted in Natural order (by GPA)  
     */  
    @Override  
    public int compareTo(Student o) {  
        // return o.gpa.compareTo(this.gpa); // hi to low  
        return this.gpa.compareTo(o.gpa); // low to hi  
    }  
    ...  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    /**  
     * Student comparators for additional specific Student Sort ordering.  
     * @param o1      Student object  
     * @param o2      Student object  
     * @return          integer result of comparison  
     */  
    public static int compareByLastName(Student o1, Student o2) {  
        return o1.getLastName().compareToIgnoreCase(o2.getLastName());  
    }  
    public static int compareByFirstName(Student o1, Student o2) {  
        return o1.getFirstName().compareToIgnoreCase(o2.getFirstName());  
    }  
    ...  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    /**  
     * Student comparators for additional specific Student Sort ordering.  
     * @param o1      Student object  
     * @param o2      Student object  
     * @return          integer result of comparison  
     */  
    ...  
    public static int compareByAge(Student o1, Student o2) {  
        return Integer.compare(o1.getAge(), o2.getAge());  
    }  
    ...  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    public static void demo() {  
        List<Student> students = new ArrayList<>();  
        students.add(new Student(44, "barack", "obama", 56, 3.2));  
        students.add(new Student(45, "donald", "trump", 71, 2.25));  
        students.add(new Student(12, "zachary", "taylor", 65, 4.0));  
        students.add(new Student(2, "john", "adams", 90, 3.0));  
        students.add(new Student(1, "george", "washington", 67, 2.5));  
        ...  
    }  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    public static void demo() {  
        ...  
        /*  
         * Sort the collection of Student objects by Natural order  
         * (Student class implements Comparable interface)  
         */  
        System.out.println(students.size()  
            + " students in the following collection: 1. SORTED BY GPA.");  
        Collections.sort(students);  
        for (Student s : students) {  
            System.out.println(s);  
        }  
        ...  
    }  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    public static void demo() {  
        ...  
        /*  
         * Sort the collection of Student objects by Natural order  
         * (Student class implements Comparable interface)  
         */  
        System.out.println(students.size()  
            + " students in the following collection: 2. SORTED BY GPA.");  
  
        // no comparator, natural order: student implements Comparable  
        students.sort(null);    students.forEach(System.out::println);  
        ...  
    }  
}
```

Sort Student Example

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    public static void demo() {  
        ...  
        // use a static comparator in class for different sort ordering  
        System.out.println(students.size() \  
            + " students in the following collection: 3. SORTED BY LAST NAME.");  
        students.sort(Student::compareByLastName);  
        students.forEach(System.out::println);  
  
        System.out.println(students.size())  
            + " students in the following collection: 4. SORTED BY FIRST NAME.");  
        students.sort(Student::compareByFirstName);  
        students.forEach(System.out::println);  
        ...  
    }  
}
```

Sort Student Example

```
public static void demo() {  
    ...  
    // use a static comparator in class for different sort ordering  
    ...  
    System.out.println(students.size()  
        + " students in the following collection: 5. SORTED BY AGE.");  
    students.sort(Student::compareByAge);  
    students.forEach(System.out::println);  
  
    System.out.println(students.size()  
        + " students in the following collection: 6. SORTED BY LASTNAME.");  
    Collections.sort(students, Student::compareByLastName);  
    students.forEach(System.out::println);  
    ...  
}  
}
```

Sort Student Example

```
public static void demo() {  
    ...  
    // use a static comparator in class for different sort ordering  
    ...  
    System.out.println(students.size()  
        + " students in the following collection: 7. SORTED BY FIRST NAME.");  
    Collections.sort(students, Student::compareByFirstName);  
    students.forEach(System.out::println);  
  
    System.out.println(students.size()  
        + " students in the following collection: 8. SORTED BY AGE.");  
    Collections.sort(students, Student::compareByAge);  
    students.forEach(System.out::println);  
}  
}
```

Sort Student Example

- CONSOLE OUTPUT

5 students in the following collection: 1. SORTED BY GPA.

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

5 students in the following collection: 2. SORTED BY GPA.

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

...

Sort Student Example

- CONSOLE OUTPUT (continued)

...

5 students in the following collection: 3. SORTED BY FIRST NAME.

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

5 students in the following collection: 4. SORTED BY LAST NAME.

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Sort Student Example

- CONSOLE OUTPUT (continued)

...

5 students in the following collection: 5. SORTED BY AGE.

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

5 students in the following collection: 6. SORTED BY FIRST NAME.

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

Sort Student Example

- CONSOLE OUTPUT (continued)

...

5 students in the following collection: 7. SORTED BY LAST NAME.

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

5 students in the following collection: 8. SORTED BY AGE.

Person: barack obama, age: 56, id: 44, is a student having a GPA of: 3.2

Person: zachary taylor, age: 65, id: 12, is a student having a GPA of: 4.0

Person: george washington, age: 67, id: 1, is a student having a GPA of: 2.5

Person: donald trump, age: 71, id: 45, is a student having a GPA of: 2.25

Person: john adams, age: 90, id: 2, is a student having a GPA of: 3.0

CSYE 6200

Concepts of Object Oriented Design

Java Exceptions

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - Java Exceptions
 - Throw
 - Try, Catch, Finally
 - Try with Resources

Exception

“An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.”

<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

Exception

- An aberration or abnormal event
 1. Exceptional Event
 2. Error or Fault
 3. Disruptive
 4. Sometimes unrecoverable

Java Exception Types

- 1. Checked Exception**
- 2. External Error or Fault**
- 3. Runtime Exception**

| Checked Exception | Error Exception | Runtime Exception |
|---|--|--|
| All exceptions, except Error, Runtime
Predictable
Must be either Caught OR Specified
Compiler Error if not caught OR Specified
<code>FileNotFoundException</code> | External Hardware Exception
Difficult to anticipate
Optional to be Caught OR Specified

<code>ErrorException & its subclasses</code>
<code>IOException</code> | Internal Exception (internal to app)
Difficult to anticipate
Optional to be Caught OR Specified

<code>RuntimeException & its subclasses</code>
<code>NullPointerException</code> |

Java Checked Exception Type

- ALL Exceptions are Checked Exceptions
 - ***EXCEPT:*** Error, RuntimeException and their subclasses.
- Predictable Error Conditions
 - File I/O with user specified invalid file name
- Must EITHER be ***Caught*** or ***Specified***
- Compiler error if neglected
- **java.io.FileReader**
 - **java.io.FileNotFoundException**

Java Error Exception Type

- Errors *External* to application
 - Hardware Error
- Difficult to anticipate
- **Error** and its subclasses
- OPTIONAL: *Caught* or *Specified*
- **java.io.IOException** Exception

Java Runtime Exception Type

- Errors *Internal* to application
 - Programming bug
- Difficult to anticipate
- **RuntimeException** and its subclasses
- OPTIONAL: *Caught* or *Specified*
- **NullPointerException** Exception

Java Custom Exception Type

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

Throw

- “When an **error** occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an ***exception object***, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called ***throwing an exception.***”
-

Throw

SYNTAX:

throw *someThrowableObject*

- Where *someThrowableObject* are instances of subclasses derived from the **java.lang.Throwable class**

Throw Example

```
public void  
throwAnException() throws Exception {  
    throw new Exception("Boo!");  
}
```

- Exception which may be thrown by method *throwAnException()* is *Specified* by using **throws Exception**

Try, Catch, Finally

SYNTAX:

```
try {  
    ...  
} catch {  
    ...  
} finally {  
    ...  
}
```

<https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html>

Try, Catch, Finally

Where:

- Try block
 - Surrounds code which *may* throw exception
- Catch block
 - Exception Handler
 - Executed *only when* exception is thrown
- Finally block
 - *OPTIONAL*
 - *Always* executed IF present

Try, Catch Example

```
try {  
    this.throwAnException();  
} catch (Exception e) {  
    System.out.println("Caught!");  
    System.out.println(e.getMessage());  
    e.printStackTrace();  
}
```

- Exception thrown by *throwAnException()* is ***Caught***
-

Try, Catch, Finally Example

```
try {  
    this.throwAnException();  
} catch (Exception e) {  
    System.out.println("Caught!");  
    System.out.println(e.getMessage());  
    e.printStackTrace();  
} finally {  
    System.out.println("Finally done!");  
}
```

Try with Resources

SYNTAX:

```
try ( open auto-closable resources ) {  
    ...  
} catch {  
    ...  
} finally {  
    ...  
}
```

AutoClosable

- Implemented by:

- BufferedReader
- FileReader
- BufferedWriter
- FileWriter
- DatagramSocket

An object that may hold resources (such as file or socket handles) until it is closed. (i.e. until the end of try {} block).

The close() method of an AutoCloseable object is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header.

This construction ensures prompt release, avoiding resource exhaustion exceptions and errors that may otherwise occur.

CSYE 6200

Concepts of Object Oriented Design

Java File Input Output

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - 1. Loops
 - 2. Array
 - 3. Map
 - 4. Scanner
 - 5. File Handling
 - 6. Exception Handling
-

Range Loop

for (Type ele : list) { }

Syntax:

for (type item : **container**)

Example:

String[] **names** =

{“Adam”, “Eve”, “Cain”, “Able”}; // 4

Strings

for (String *name* : **names**) {

 System.out.println(*name*); // executed 4

times

}

Range Loop Example

```
List<String> col = new ArrayList<String>();  
  
col.add("Peter");  
col.add("Paul");  
col.add("Mary");  
  
for (String item : col) {  
    System.out.print(item.toUpperCase() +", ");  
}
```

Produces Output:

PETER, PAUL, MARY,

Iterator Loop

Syntax:

```
Iterator<Integer> it;
```

Example:

```
List<Integer>numbers = new Vector<>();
```

```
Iterator<Integer> it = numbers.iterator();
```

- Use with collections supporting Iterator interface
- Iterator type is specific to type of collection (*Vector*) and type of it's contents (*Integer*)

Iterator Loop

```
List<String> names = new Vector<String>();  
Iterator<String> it = names.iterator()  
while (it.hasNext()) {  
    System.out.println(it.next());  
}
```

Iterator Loop Example

```
List<Integer> numbers = new ArrayList<Integer>();  
numbers.add(1); // auto-boxing  
numbers.add(2); // same as  
numbers.add(3); // numbers.add(new Integer(3));
```

```
Iterator<Integer> it = numbers.iterator();  
while (it.hasNext()) {  
    System.out.print(it.next() +', ', );  
}
```

Produces Output:

1, 2, 3,

Iterator Loop Example

```
List<String> names= new ArrayList<String>();  
names.add("Peter");  
names.add("Paul");  
names.add("Mary");
```

```
Iterator<String> it = names.iterator();  
while (it.hasNext()) {  
    System.out.print(it.next() +", ");  
}
```

Produces Output:

Peter, Paul, Mary,

ListIterator Loop

Syntax:

```
ListIterator<Integer> it;
```

Example:

```
List<Integer> numbers = new Vector<>();
```

```
ListIterator<Integer> it = numbers.listIterator();
```

- Use with collections supporting ListIterator interface
- Can iterate forward and backwards
- Can modify element (set)

ListIterator Loop

```
List<String> names = new Vector<String>();  
ListIterator <String> it = names.listIterator()  
while (it.hasNext()) {  
    System.out.println(it.next());  
}  
  
while (it.hasPrevious()) {  
    System.out.println(it.previous());  
}
```

ListIterator Loop Example

```
List<String> names= new ArrayList<String>();  
names.add("Peter");  
names.add("Paul");  
names.add("Mary");
```

```
ListIterator<String> it = names.listIterator();  
while (it.hasNext()) {  
    String item = it.next();  
    item = item.toUpperCase() + ", ";  
    it.set(item); // only in ListIterator  
    System.out.print(it2.next() +", ");  
}
```

Produces Output:

PETER, PAUL, MARY,

ListIterator Loop Example

```
List<String> numbers= new ArrayList<String>();  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);
```

```
ListIterator<Integer> it = numbers.listIterator();  
while (it.hasNext()) {  
    System.out.print(it.next() +", ");  
}
```

Produces Output:

1,2,3,

Array

Syntax:

```
String[] names = {"adam", "eve"};
```

```
int[] numbers = { 1, 2, 3 };
```

```
String[] strArray = new String[3];
```

- contains multiple items of same type
- Items in contiguous memory
- Fixed size
- Random access
- Array is an object (Reference Type)

Array Examples

```
int[] numbers = { 1,2,3 };

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

Produces Output:

1, 2, 3,

Array Examples

```
String[] names = {"adam", "eve"};
```

```
for (String name : names) {  
    System.out.print(name + ", ");  
}
```

Produces Output:

adam, eve,

Array Examples

```
String[] threeFruit = {"Apple", "Pear", "Orange"};
```

```
for (String fruit : threeFruit) {  
    System.out.print(fruit +', ');  
}
```

- Produces Output:
Apple, Pear, Orange,

Associative

Map

Syntax:

Map<KeyType, ValueType> m;

eg. **Map<Integer, String>** m =

new **HashMap<Integer, String>()**;

- Interface
- Implemented by HashMap class

Map Examples

```
Map<String, String> states = new HashMap<String, String>();
```

```
states.put("NH", "New Hampshire");
```

```
states.put("NJ", "New Jersey");
```

```
states.put("NY", "New York");
```

```
System.out.println("NY: " + states.get("NY"));
```

```
System.out.println("NJ: " + states.get("NJ"));
```

Produces Output;

NY: New York

NJ: New Jersey

Scanner

- Text scanner
- Parses tokens using delimiter
- Converts to primitive types and Strings

Scanner Examples

```
Student dan = new Student();
```

```
Scanner in = new Scanner("peters,dan,17,3.25,james,peters");
in.useDelimiter(",");
```

```
dan.setLname(in.next()); // no conversion, String token is String
dan.setFname(in.next());
dan.setAge(in.nextInt()); // convert String token to int
dan.setGpa(in.nextDouble()); // convert String token to double
dan.setParentFname(in.next());
dan.setParentLname(in.next());
in.close();
students.add(dan); // add Student object to students container
```

Scanner Examples

```
// show collection of students  
System.out.println(students.size() + " Students in roster.");  
for (Student student : students) {  
    System.out.println(student.getLname()  
        + ", " + student.getFname()  
        + ", Age " + student.getAge()  
        + ", GPA: " + student.getGpa());  
}
```

- Produces Output

1 Students in roster.

peters, dan, Age 17, GPA: 3.25

Scanner Examples

```
// show collection of students  
System.out.println(students.size() + " Students in roster.");  
for (Student student : students) {  
    System.out.println(student); // Student toString()  
}
```

- Produces Output

1 Students in roster.

peters, dan, Age 17, GPA: 3.25

BufferedReader

Syntax:

```
BufferedReader in = new BufferedReader( new  
FileReader("inputFile.txt"));  
  
String thisLine = in.readLine();  
  
in.close();
```

- BufferedReader uses FileReader
- Adds efficiency by buffering data before calling FileReader

Buffered Reader Example

```
String thisLine = null;  
  
// try with resources: all resources in () are closed at conclusion of try clause  
try ( // open input stream from input file for reading purpose.  
        FileReader fr = new FileReader(fileName);  
        BufferedReader in = new BufferedReader(fr); ) {  
    System.out.println("BufferedReader: '" + this.inputFileName + "'");  
    while ((thisLine = in.readLine()) != null) {  
        System.out.println(thisLine);  
    }  
} catch(Exception e){  
    e.printStackTrace();  
}
```

Try with Resources

- Use with resources (i.e. file descriptors used for file I/O)
 - **MUST** Instantiate class **INSIDE** of try parenthesis
 - `try (... new...)`
- Class **MUST** implement **AutoCloseable** Interface
 - Will be automatically closed at conclusion of `try { ... }`

```
try (      FileReader fr = new FileReader(fileName);
          BufferedReader in = new BufferedReader(fr);  ) {
    ...
} catch(Exception e){
    e.printStackTrace();
}
```

BufferedWriter

Syntax:

```
FileWriter fw = new FileWriter ("outputFile.txt");
```

```
BufferedWriter bw = new BufferedWriter (fw);
```

```
PrintWriter pr = new PrintWriter (bw);
```

- BufferedWriter uses a FileWriter object
- Newline is system specific
 - bw.newLine()
- flush after write
 - bw.flush();

Buffered Writer Example

```
String[] fiveNames = {"Dan", "Jim", "Eve", "Ina"};  
  
// try with resources: all resources in () are closed at conclusion of try clause  
try (          // open output stream to output file for writing purpose.  
      FileWriter fw = new FileWriter(fileName);  
      BufferedWriter out= new BufferedWriter(fw);  
) {  
    System.out.println("BufferedWriter: '" + this.outputFileName + "", write '' + fiveNames.length + "  
items");  
    for (String name : fiveNames) {  
        out.write(name);  
        out.newLine();  
    }  
    out.flush();  
} catch (Exception e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

Buffered Writer Example

```
// try with resources:  
// all resources in () are closed at conclusion of try clause  
try ( // open output stream to output file for writing.  
    FileWriter fw = new FileWriter(fileName);  
    BufferedWriter out= new BufferedWriter(fw);  
) {  
    for (String name : fiveNames) {  
        out.write(name);  
        out.newLine();  
    }  
    out.flush();
```

Buffered Writer Example

```
} catch (Exception e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

Read CSV Example 1

```
List<Student> students = new ArrayList<Student>();  
// Read entire ASCII comma separated value file: each line becomes a student object  
// try with resources: all resources in () are closed at conclusion of try clause  
try (BufferedReader inLine = new BufferedReader(new FileReader(fileName));  
) {  
    String inputLine = null;      // read one line from file at a time  
    while ((inputLine = inLine.readLine()) != null) {  
        // Parse line converting each string token into a Student object field  
        String[] fields = inputLine.split(",");  
        String lname = fields[0];  
        String fname = fields[1];  
        int age = new Integer(fields[2]);  
        double gpa = new Double(fields[3]);  
        String parentFname = fields[4];  
        String parentLname = fields[5];  
  
        // instantiate Student object from line in file and add to list  
        students.add(new Student (fname,lname,age,gpa,parentFname,parentLname));  
    }  
} catch (IOException e) {  
    // catch IOException (and implicitly FileNotFoundException)  
    e.printStackTrace();  
}
```

Read CSV Example 1

```
List<Student> students = new ArrayList<Student>();  
  
// Read entire ASCII comma separated value file:  
// each line becomes a student object  
// try with resources:  
// all resources in () are closed at conclusion of try clause  
try (BufferedReader inLine = new BufferedReader(new  
FileReader(fileName));  
) {  
    ...  
}
```

Read CSV Example 1

```
) {  
  
// read one line from file at a time  
    String inputLine = null;  
    while ((inputLine = inLine.readLine()) != null) {  
        // Parse line converting each string token  
        // into a Student object field  
        String[] fields = inputLine.split(",");  
        String lname = fields[0];  
        ...  
    } catch (IOException e) {
```

Read CSV Example 1

...

```
while ((inputLine = inLine.readLine()) != null) {
```

```
    // Parse line converting each string token into a Student  
    object field
```

```
    String[] fields = inputLine.split(",");
```

```
    String lname = fields[0];
```

```
    String fname = fields[1];
```

```
    int age = new Integer(fields[2]); // see *
```

```
    double gpa = new Double(fields[3]); // see *
```

```
    String parentFname = fields[4];
```

```
    String parentLname = fields[5];
```

...

* String type conversion

- * Conversion from String type to int requires exception handling
 - Catch exception thrown from failure to convert bad String

...

```
int age = 0;  
try {  
    age = new Integer(fields[2]);  
} catch (NumberFormatException e) {  
    age = 0;  
    e.printStackTrace();  
}
```

...

* String type conversion

- * Conversion from String type to double requires exception handling
 - Catch exception thrown from failure to convert bad String

...

```
double gpa = 0;  
try {  
    gpa = new Double(fields[3]);  
} catch (NumberFormatException e) {  
    gpa = 0.0;  
    e.printStackTrace();  
}
```

...

Read CSV Example 1

...

```
// instantiate a Student object from EACH line
// in file and add to list
students.add(new Student ( fname,
lname,
age,
gpa,
parentFname,
parentLname ) );

} // end while loop
```

...

Read CSV Example 1

```
...  
} catch (IOException e) {  
    // catch IOException (and implicitly  
    FileNotFoundException)  
    e.printStackTrace();  
}
```

Read CSV Example 2

```
List<Student> students = new ArrayList<Student>();  
try {  
    Scanner inLine = new Scanner(new BufferedReader(new FileReader(fileName)));  
  
    while (inLine.hasNextLine()) {  
        String inputLine = inLine.nextLine();  
        Scanner in = new Scanner(inputLine);  
        in.useDelimiter(",");  
        String lname = in.next();  
        String fname = in.next();  
        int age = in.nextInt();  
        double gpa = in.nextDouble();  
        String parentFname = in.next();  
        String parentLname = in.next();  
  
        students.add(new Student (fname,lname,age,gpa,parentFname,parentLname));  
        in.close();  
    }  
    inLine.close();  
} catch (FileNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

Read CSV Example 2

```
List<Student> students = new ArrayList<Student>();  
try {  
    // Scanner used to read each line from file  
    Scanner inLine = new Scanner(new  
        BufferedReader(new FileReader(fileName)));  
  
    while (inLine.hasNextLine()) {  
        String inputLine = inLine.nextLine();  
        ...  
    }  
    inLine.close(); // scanner resource must be closed  
} catch (FileNotFoundException e) {  
    ...  
}
```

Read CSV Example 2

```
while (inLine.hasNextLine) {  
    String inputLine = inLine.nextLine();  
    // Scanner ALSO used to read each token in line  
    Scanner in = new Scanner(inputLine);  
    in.useDelimiter(",");  
    String lname = in.next();  
    String fname = in.next();  
    int age = in.nextInt(); // See *  
    double gpa = in.nextDouble(); // See *  
    String parentFname = in.next();  
    String parentLname = in.next();  
  
    ...  
}
```

Read CSV Example 2

```
while (inLine.hasNextLine()) {  
    String inputLine = inLine.nextLine();  
  
    Scanner in = new Scanner(inputLine);  
    in.useDelimiter(","); // CSV file uses comma  
    String lname = in.next();  
  
    ...  
  
    students.add(new Student  
(fname,lname,age,gpa,parentFname,parentLname));  
    in.close(); // must close scanner  
}  
  
inLine.close(); // must close scanner  
} catch (FileNotFoundException e) {  
    ...  
}
```

Read CSV Example 2

```
} catch (FileNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

CSYE 6200

Concepts of Object Oriented Design

Packaging

Daniel Peters

d.peters@neu.edu

-
- Lecture:
 1. Java Packages
 2. Java 9 Modules
 3. Java Jar Files
-

Java Packages

java.lang.String

“ A package is a namespace that organizes a set of related classes and interfaces. “

- Online Information

<https://docs.oracle.com/javase/tutorial/java/concepts/package.html>

Java Packages

“... Enormous class library (a set of packages)...”

- Online Information

<https://docs.oracle.com/javase/tutorial/java/concepts/package.html>

Java Packages

“**The Java Platform API Specification** contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform.”

- Online Information

<https://docs.oracle.com/javase/tutorial/java/concepts/package.html>

Java Packages

“ To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages. ”

– Online Information

<https://docs.oracle.com/javase/tutorial/java/package/packages.html>

Java Packages

“ **Definition:** A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes and interfaces*. ”

– Online Information

<https://docs.oracle.com/javase/tutorial/java/package/packages.html>

Java Packages

“ The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your types in packages too. ”

– Online Information

<https://docs.oracle.com/javase/tutorial/java/package/packages.html>

Java Packages

“ The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your types in packages too. ”

– Online Information

<https://docs.oracle.com/javase/tutorial/java/package/packages.html>

Java Packages

“ So far, most of the examples in this tutorial have referred to types by their simple names, such as Rectangle and StackOfInts. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported. ”

– Online Information

<https://docs.oracle.com/javase/tutorial/java/package/packages.html>

Java Packages

“ However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. ”

e.g., `graphics.Rectangle`

– Online Information

<https://docs.oracle.com/javase/tutorial/java/packages.html>

Java Packages

“ To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Rectangle class from the graphics package created in the previous section. ”

e.g., `import graphics.Rectangle;`

– Online Information

<https://docs.oracle.com/javase/tutorial/java/packages.html>

Java Packages

“ To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character. ”

e.g., `import graphics.*;`

– Online Information

<https://docs.oracle.com/javase/tutorial/java/packages.html>

Java 9 Modules

“Modularity adds a higher level of aggregation above packages.”

“The key new language element is the *module*—a uniquely named, reusable group of related packages,...”

- Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“As we mentioned, a module must provide a module descriptor—metadata that specifies the module’s dependencies, the packages the module makes available to other modules, and more.”

– Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“A module descriptor is the compiled version of a module declaration that’s defined in a file named ‘***module_info.java***’ ”

– Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“ Each module declaration begins with the keyword `module`, followed by a unique module name and a module body enclosed in braces, as in: ”

```
module modulename {  
}
```

– Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“The module declaration’s body can be empty”

```
module project1 {  
}
```

– Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“or may contain various *module directives*, including”

- Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

- Module directives
 - Requires
 - Exports
 - Provides...with
 - Uses
 - Opens
 - Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

“**requires**. A `requires` module directive specifies that this module depends on another module—this relationship is called a *module dependency*. Each module must explicitly state its dependencies.”

```
module project1 {  
    requires module;  
}
```

– Online Information

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

Java 9 Modules

Module `java.desktop`

“ Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans. ”

```
module project1 {  
    requires java.desktop;  
}
```

– Online Information

<https://docs.oracle.com/javase/9/docs/api/java.desktop-summary.html>

Java JAR Files

java.lang.String

- Java Archive

- Zip archive file format
- Use Java Archive tool (jar) in Java Development Kit (JDK)
- Online Information
 - <https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html>

Java JAR Tool

- Create a jar file

jar cf *jar-file* *input-file(s)*

- c: create
- f: name of jar file for tool output

- Example:

jar cf mydriver.jar DriverProject.class

Eclipse JAR Tool

1. Select Project in Eclipse Project Explorer
2. Select menu: File > Export
3. Select Export Destination
 - a. Type jar
 - b. Select Runnable JAR or JAR file
 - c. Select Next

Eclipse JAR Tool (cont'd)

4. Select Options or accept defaults
5. JAR file:
 - a. Browse to directory
 - b. Type jar file name, e.g. myjar.jar
6. Select FINISH

Java JAR Tool on Windows

1. Copy java classes to single folder
2. Start cmd prompt window as admin
3. Change to folder containing java classes
4. Insure java JDK bn directory is in path:
 - a. C:\Program Files\Java\jdkn.n.nn\bin
5. Create jar from all java classes
 - a. jar cf myjar.jar c1.class c2.class ... cn.class

Java JAR Tool

- View contents of a jar file

`jar tf jar-file input-file(s)`

- t: view table of contents
- f: name of jar file for tool output

- Example:

`jar tf mydriver.jar DriverProject.class`

Java command

- Run a class (which contains a main method)

`java classNameWithout.classExtension`

- Example

- Working directory: `./Proj/bin`
 - Class file: `./Proj/bin/c1.class`
 - Package: (class in default package):

`java c1`

Java command

- Run a class (which contains a main method)

`java classNameWithout.classExtension`

- Example

- Working directory: `./Proj/`
 - Class file: `./Proj/bin/edu/neu/java1/c1.class`
 - Package: `edu.neu.java1`

`java -cp bin edu.neu.java1.c1`

-

Java command

- Run an executable jar file

`java -jar jar-file`

- Example:

`java –jar mydriver.jar`

Java JAR Tool

- Create a jar file with specific entry point

`jar cfe jar-file input-file(s)`

- c: create
 - f: name of jar file for tool output
 - e: entry point (class with main method)
- Example:

`jar cfe mydriver.jar c1.class c2.class c1.class`

CSYE 6200

Concepts of Object Oriented Design

Interface and Inner Class

Daniel Peters

d.peters@neu.edu

-
- Lecture
 1. Java Inner Interface
 2. Java Inner Class
 3. Java Interface
-

Private Inner Interface

```
public class OuterClass {  
    ...  
    private interface InnerHelperInterface {  
        ...  
    }  
    ...  
}
```

Public Inner Interface

```
public class OuterClass {  
    ...  
    public interface InnerRelatedInterface {  
        ...  
    }  
    ...  
}
```

Inner Interface

- Inner Interface
 - CAN NOT be defined in an inner class.
 - IS An Outer Class member (like member data)
 - Public, Private or Protected
 - Controls outside visibility to inner class
 - NOTE: outer class is ALWAYS public
 - Static: **implicit** (and redundant) for Java Interfaces
(like all Interface methods are implicitly public and public access modifier is redundant)

Private Inner Class

```
public class OuterClass {  
    ...  
    private class InnerHelperClass {  
        ...  
    }  
    ...  
}
```

Public Inner Class

```
public class OuterClass {  
    ...  
    public class InnerRelatedClass {  
        ...  
    }  
    ...  
}
```

Inner Class

- Inner Class
 - IS An Outer Class member (like member data)
 - Public, Private or Protected
 - Controls outside visibility to inner class
 - NOTE: outer class is ALWAYS public
 - Static: inner class belongs to Class
 - Program scope: always exists
 - Non-static: belongs to each object instance
 - Object instance scope: doesn't exist until object creation
 - Must be instantiated from an object instance (**outerClass**)

```
OuterClass outerClass = new OuterClass(); // object
InnerClass innerClass = outerClass.new InnerClass();
```

Inner Class

- Inner Class
 - Methods:
 - Static: requires inner class to ALSO be static
 - Non-static: belongs to each object instance

Anonymous Inner Class

- Anonymous Inner Class
 - In-Line unnamed class specification
 - Supplied to method as a parameter
 - Usually an implementation of an Interface

Anonymous Inner Class

```
List<Student> students = new ArrayList<>();
```

```
students.sort(new Comparator<Student>() {
```

```
    @Override  
    public int compare(Student o1, Student o2) {  
        return o1.getAge().compareTo(o2.getAge());  
    }  
});
```

- Sort requires a Comparator object
-

Lambda Expression

```
List<Student> students = new ArrayList<>();  
  
students.sort( (o1, o2) -> o1.getAge().compareTo  
(o2.getAge() );  
}  
});
```

- Sort requires a Comparator object

Java AutoBoxing

“*Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.”

<https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>

Java AutoBoxing

- AutoBoxing
 - Character ch = 'a';
 - Double n = 2.2;
 - Integer i = 3;

Integer class and Primitive int

- AutoBoxing int literal values with Integer class

```
List<Integer> numbers = new  
ArrayList<>(Arrays.asList(1,3,2));
```

- UnBoxing int from List of Integer objects

```
int i = numbers.get(numbers);
```

Integer class and Primitive int

- Integer number = 7;
 1. Reference type
 2. Class must be instantiated or set to null
 3. Object has helpful members
 4. Requires additional storage

Integer class and Primitive int

- `int n = 7;`
 1. Primitive type
 2. Does NOT require instantiation
 3. Does NOT have any members
 4. Requires minimal storage
 5. Fast access

Java Interface

- Java Interface
 - Reference type, similar to a class
 - Contains ONLY:
 - Data
 - Public Static Final (Constants)
 - Methods:
 - ALL METHODS IMPLICITLY PUBLIC
 - » Keyword public may be omitted from methods.
 - abstract methods (no body)
 - Static methods
 - Default methods (Java 8)

Java Interface

- Java Interface
 - Public:
 - Public interface is accessible to all classes
 - Otherwise, only accessible to classes in same package
 - Keyword interface
 - Must be implemented by a class
 - Class MUST implement all interface methods
 - Can be extended by other interfaces

Java Interface

```
public interface GroupedInterface extends Interface1,  
Interface2, Interface3 {  
    // constant declarations  
    // base of natural logarithms  
    double E = 2.718282;  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

<https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html>

Java Interface

public interface GroupedInterface

- Public: interface available to all classes
- Private: interface available to package ONLY
- Keyword interface required
- Keyword extends:
 - An interface can use multiple other interfaces
- ALL methods are public
 - Keyword ‘public’ optional for all methods
 - Abstract methods are signature only
 - Default and Static methods with implementations

Java Interface

- Java Interface Use
 - Must be implemented by a class
 - Class MUST implement all interface methods
 - Can be extended by another interface

Java Interface

- Java Interface as API
- “The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. **APIs are also common in commercial software products.** Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's **API is made public (to its customers), its implementation of the API is kept as a closely guarded secret**—in fact, it **may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.**”

<https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>

Java Interface

- Java Interface Examples
 - 1. Comparable
 - 1. Implement to make your class sortable by default in natural order.
 - 2. Comparator
 - 1. Implement and use to specify a specific sort order.
 - 3. Runnable
 - 1. Implement to make your class executable on a new thread.

Comparable Interface

- Sort Pizza objects by pizza price
 - Comparable Interface provides a natural order

```
public class Pizza implements Comparable<Pizza> {  
    ...  
    public int compareTo (Pizza o) {  
        return Double.compare(this.getPrice(), o.getPrice());  
    }  
}  
List<Pizza> pizzaList = new ArrayList<>();  
Collections.sort(pizzaList);      // Comparable (natural) order
```

Comparable Interface

```
public class Student extends Person implements Comparable<Student> {  
    . . .  
    public int compareTo ( Student o) {  
        return Double.compare(this.getGpa(), o.getGpa());  
    }  
}  
  
List< Student > students = new ArrayList<>();  
Collections.sort(students);    // Comparable (natural) order
```

Comparator Interface

```
public class CompareByPrice implements  
Comparator<Price> {  
  
    @Override  
    public int compare(Pizza o1, Pizza o2) {  
        return Double.compare(o.getPrice(), o.getPrice());  
    }  
}  
  
List<Pizza> pizzaList = new ArrayList<>();  
pizzaList.sort(new CompareByPrice());      // Comparator  
order
```

Comparator Interface

```
public class CGPA implements Comparator<Student> {  
  
    @Override  
    public int compare(Student o1, Student o2) {  
        return Double.compare(o1.getGpa(), o2.getGpa());  
    }  
}  
  
List< Student > students = new ArrayList<>();  
students.sort(new CGPA()); // Comparable (natural) order
```

Comparator Interface

```
public class Student extends Person implements Comparable < Student > {  
    ...  
    private static class Ranking implements Comparator<Student> {  
        @Override  
        public int compare(Student o1, Student o2) {  
            // sort by GPA High to Low  
            return Double.compare(o2.getGpa(), o1.getGpa());  
        }  
    }  
    public static void ranking(List<Student> students) {  
        students.sort(new Ranking());  
    }  
    ...  
    List< Student > students = new ArrayList<>();  
    Student.ranking(students); // Comparator GPA order
```

Comparator Interface

```
public class Student extends Person implements Comparable<Student> {  
    ...  
    private static class Ranking implements  
        Comparator<Student> {  
        @Override  
            public int compare(Student o1, Student o2) {  
                // sort by GPA High to Low  
                return Double.compare(o2.getGpa(), o1.getGpa());  
            }  
    }  
    ...  
}
```

Comparator Interface

- Class implements a static sorting method

```
public class Student extends Person implements Comparable<Student>
{
    ...
    public static void ranking(List<Student> students) {
        students.sort(new Ranking());
    }
}
```

Comparator Interface

- Use the static class sorting method

```
List< Student > students = new ArrayList<>();  
Student.ranking(students);      // Comparator GPA order
```

CSYE 6200

Concepts of Object Oriented Design

SOLID

Daniel Peters

d.peters@neu.edu

-
- Lecture **S O L I D:**
 - Single Responsibility Principle:
 - Open-Closed Principle:
 - Liskov Substitution Principle:
 - Interface Segregation Principle:
 - Dependency Inversion Principle:

SOLID

- Acronym coined by Michael Feathers
- Design principles promoted by Robert C. Martin.
 - Benefits:
 - Flexibility
 - Maintainability
 - Understandability
- <https://en.wikipedia.org/wiki/SOLID>

Single Responsibility Principle:

- Design each class (or module) with **ONLY** one **Single** responsibility (*purpose* or task);
- Employ Encapsulation
 - 1. All data (supporting *purpose*) class private;
 - 2. Supply public API (supporting *purpose*);

Single Responsibility Principle:

```
public class Person {  
    private int age = 17;  
    private String name = "Dan";  
    public String toString() {  
        return name + ", age " + age;  
    }  
}
```

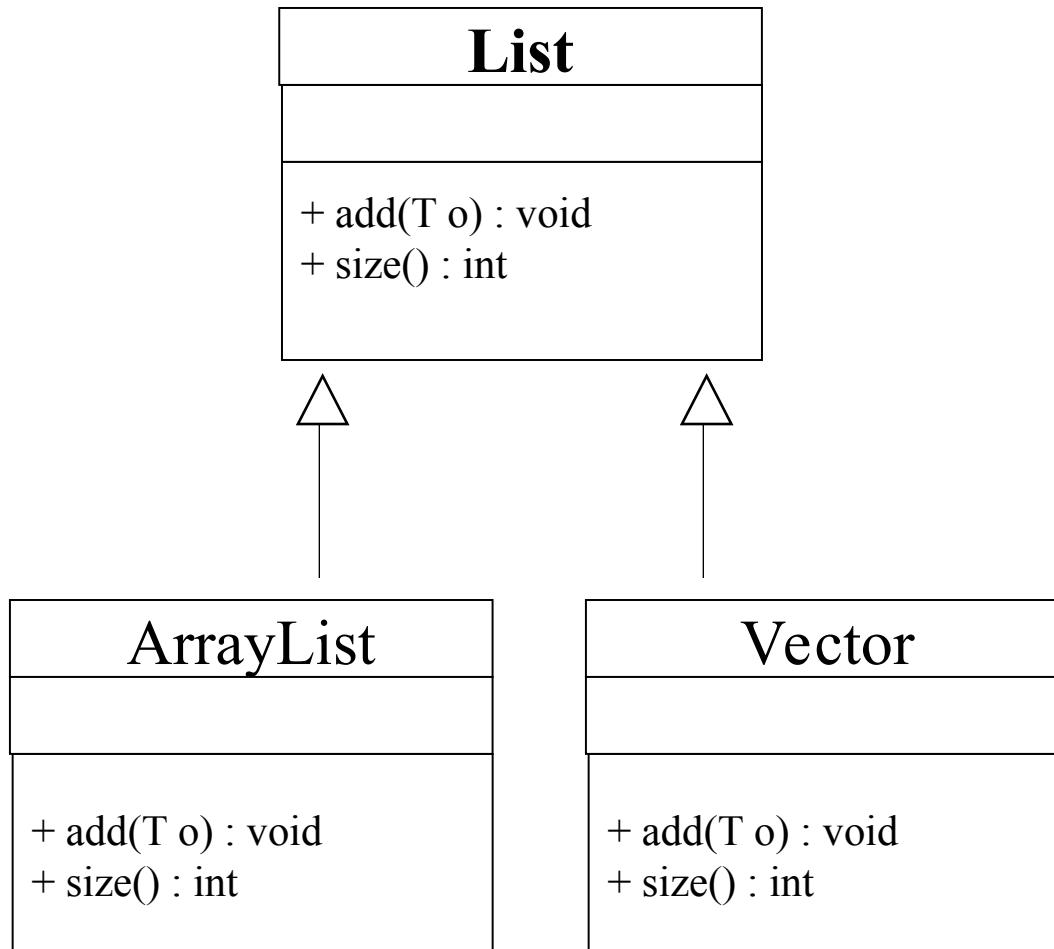
NOT Single Responsibility Principle:

```
public class Person {  
    private int age = 17;  
    private String name = "Dan";  
    private double gpa = 4.0;  
    private double wage = 25.75;  
    private String toString() {  
        return name+" "+gpa+" $ "+wage+{/hour  
        age "+ age +";  
    }  
}
```

Open-Closed Principle:

- Design and each class is **Open** to extension;
 - Use NEW subclasses and Polymorphism
- Each class is **Closed** to modifications;
 - Once tested, use class as a Super class

List Class Diagram



Open-Closed Principle:

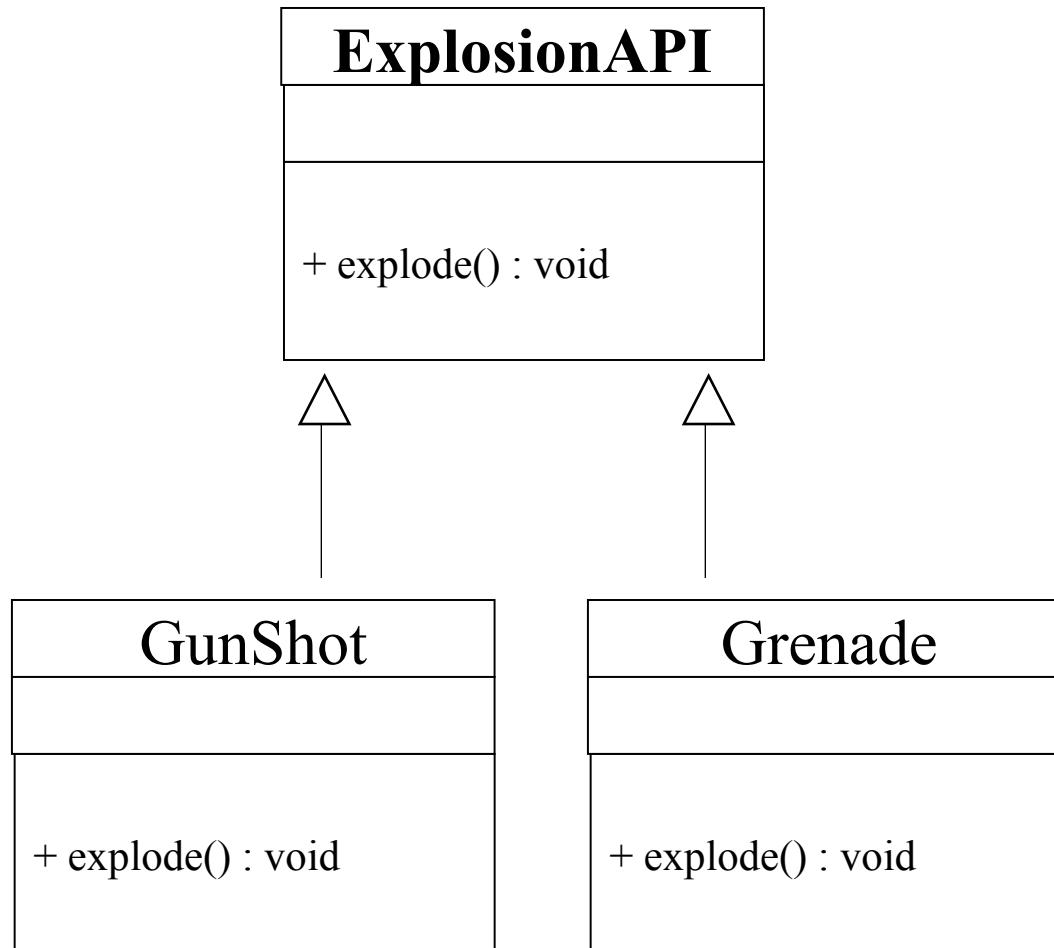
- use derived objects through superclass API

```
List<String> names = new ArrayList<>();
```

- NOT: use derived objects explicitly

```
ArrayList<String> names = new  
    ArrayList<>();
```

Explosion Class Diagram



Liskov Substitution Principle:

- IS-A Relationship: Any subclass may be **Substituted** for its Super class;
- Run-time Polymorphism;
- Employ Strong subtyping by implementing specific interfaces in subclass;
 - Can be used to differentiate subclasses;

Liskov Substitution Principle:

- use derived objects in substitution of superclass API

```
List<String> names = new ArrayList<>();
```

```
List<String> names = new Vector<>();
```

```
List<String> names = new LinkedList<>();
```

Interface Segregation Principle:

- No class should depend on any method it does not implement (use);
 - Design Interface:
 - Small granularity
 - fine grained like Salt, not Snowballs
 - Less IS More: Few methods in interface;
 - Very focused and Specific purpose;
 - Class implements multiple interfaces for desired functionality;

Interface Segregation Principle:

<http://developer.classpath.org/doc/java/lang/Runnable-source.html>

```
public interface Runnable {  
    void run();  
}
```

<http://developer.classpath.org/doc/java/lang/Comparable-source.html>

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

Dependency Inversion Principle:

- Loosely coupled Design;
- Depend on abstractions.
 - Functionality Hiding: Subclass is never named
 - Don't Explicitly call new or static methods

Dependency Inversion Principle:

- Class ExplosionController should use Class Explosion
 - Class GunShot extends Explosion
 - Class Grenade extends Explosion

Dependency Inversion Principle:

- Depend on List API as abstraction

```
List<String> names = new ArrayList<>();
```

- Use of Factory design pattern abstracts **new** and completes the abstraction of derived classes

- NOT: use derived objects explicitly

```
ArrayList<String> names = new  
    ArrayList<>();
```

SOLID and OOP

- Loose Coupling is achieved by:
 - Object Oriented Principles (OOP): AIP
 - Abstraction
 - Inheritance
 - Polymorphism
 - SOLID principles: OLD
 - Open-Closed Principle
 - Liskov Substitution Principles
 - Dependency Inversion Principles

CSYE 6200

Concepts of Object-Oriented Design

Java GoF Singleton Design Pattern

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - Singleton pattern

Design Patterns

- Gang of Four (GoF) Book

Design Patterns

Elements of Reusable Object-Oriented Software
by Erich Gamma, Richard Helm, Ralph Johnson
John Vlissides

ISBN-10: 0201633612

ISBN-13: 9780201633610

Singleton Design Pattern

- Singleton Pattern is introduced in GoF Design Patterns Book:
 - Ensure a class has **ONLY ONE SINGLE instance and provide a GLOBAL point of access to it**"
- Abstraction: Functionality Hiding
 - Abstracts Class Constructor
- Used for Logging, Configuration, Factories
- **Creational Design Pattern**

Singleton Design Pattern

- Anatomy of a Design Pattern
 1. Who (are): Constituent Components (parts)
 2. What (happens): Operational
 3. Where (useful): Scenarios
 4. Why (design) Rationale
 5. When (used): Benefits

Singleton Design Pattern

- Who:

Constituent components, roles or parts participating in the deployment and use of the design pattern

1. Get Instance method

- Allows clients global access to one unique object instance of the Singleton class

2. May create its own one and only unique instance

Singleton Design Pattern

- What:

A brief description of what happens with the use of the design pattern

- 1.Client uses Get Instance method to obtain the one and only unique object instance of the Singleton class

Singleton Design Pattern

- Where:

A list of suitable scenarios for the use of the design pattern

- Where only one instance of a class is required which must be globally accessible to clients
 - 1.A Factory class requires only a single instance
 - 2.Unique Subsystem facilities, i.e., Logging, Print Spooler, GUI Window Manager, Configuration settings, require only a single instance

Singleton Design Pattern

- Why:

A brief description of rationale behind the design of the design pattern

1. Some classes require only a single instance and a global accessibility to that instance

Singleton Design Pattern

- When:

A brief description of the benefits obtained when using the design pattern

- 1.Requirement for exactly one object instance of a class.
- 2.Requirement for Global accessibility of that instance.

Singleton Design Pattern

- Allows for easy extension of design
 - Can extend or replace created object with a new class
 - S.O.L.I.D. principles (O.L.D.) saves re-test
 - Never touch existing code once tested and deployed

Singleton Design Pattern

- **Private** constructor is never used by caller
- Public **getInstance()** method returns the one and only instance of singleton class
- Singleton class ensures that one and **ONLY one instance** of itself is instantiated

Singleton Design Pattern

- Serves as a ‘better’ global variable
 - Accessible like a global
 - Guarantees ONLY SINGLE INSTANCE
- **ALL GLOBALS ARE BAD**
 - Bad for multithreaded implementations
 - Global access is problematic

Singleton Design Pattern

- Singleton useful for subsystems which **MUST** have one common instance
 - Object instantiation Factories
 - System Printer spooler
 - System Logging facility
 - System Configuration facility
 - System Graphic Window Manager
 - A/D converter for a digital filter
 - Company's official accounting system

Singleton Design Pattern

- Alternative to Singleton: Static class operations (methods) insure a single instance BUT...
 - No Loose Coupling
 - Explicitly used
 - Without S.O.L.I.D. principles

Singleton Design Pattern

- Singleton can be modified to variable number of instances
 - Change design to allow for more than a single instance
 - Control the number of application instances used

Eager Singleton Example

```
public class EagerSingleton {  
    private static final EagerSingleton instance =  
        new EagerSingleton();  
  
    private EagerSingleton() {  
    }  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

Simple Singleton

- Early or Eager initialization
 - 1. Resource is always available
 - Allocated From beginning of program execution
 - 2. Resource immediately available without delay
 - Real-Time Performance Benefit
 - 3. Resource is allocated EVEN IF NEVER USED
 - Can be a waste resources

Lazy Singleton Design Pattern

```
public class LazySingleton {  
    private static LazySingleton instance;  
    private LazySingleton() {  
        instance = null;  
    }  
    public static synchronized LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

Lazy Singleton

- **Late or Lazy** initialization
 - 1. Resource allocated WHEN REQUESTED
 - Preserves resources until actually required
 - 2. Resource NOT immediately available
 - Allocation delay can be a Real-Time Performance Hit

Singleton Example

```
public class OnlyOne {  
    private static OnlyOne instance; // data member  
    private OnlyOne() {  
        instance = null;  
    }  
    public static synchronized OnlyOne getInstance() {  
        if (null == instance) {  
            instance = new OnlyOne();  
        }  
        return instance;  
    }  
}
```

Singleton: Private Constructor

```
public class OnlyOne {  
    private static OnlyOne instance;  
    private OnlyOne() {  
        instance = null;  
    }  
    . . .  
}
```

Singleton: Public API

```
public class OnlyOne {  
    private static OnlyOne instance; // data member  
    ...  
    public static synchronized OnlyOne getInstance() {  
        if (null == instance) {  
            instance = new OnlyOne();  
        }  
        return instance;  
    }  
}
```

Singleton design Pattern Usage

```
OnlyOne singletonInstance = null;  
singletonInstace = OnlyOne.getInstance();  
singletonInstance.toString();
```

Singleton Design Pattern Summary

- Eager or Early Initialization:
 - Resource allocated at program start before run.
 - Waste of resource if never actually requested.
- Lazy or Late Initialization:
 - Resource allocated at run-time when requested.
 - Conserves Resource until actually required.

CSYE 6200

Concepts of Object-Oriented Design

Java GoF Factory Design Pattern

Daniel Peters

d.peters@neu.edu

-
- Lecture
 - Simple Factory Pattern
 - Factory Method Pattern

Design Patterns

- Gang of Four (GoF) Book

Design Patterns

Elements of Reusable Object-Oriented Software
by Erich Gamma, Richard Helm, Ralph Johnson
John Vlissides

ISBN-10: 0201633612

ISBN-13: 9780201633610

Simple Factory Pattern

- Creational Design Pattern
 - Abstraction: Functionality Hiding
 - Abstracts Class Constructor
- Employs Inheritance
 - Superclass as API
 - Interface
 - Abstract Class
- Single Factory class creates an instance of a **concrete derived target class**
 - Return instantiated **target object** to caller

Simple Factory Pattern Components

1. Design ONE target Superclass

- API to Abstract all derived classes
 - Example: **Explosion**

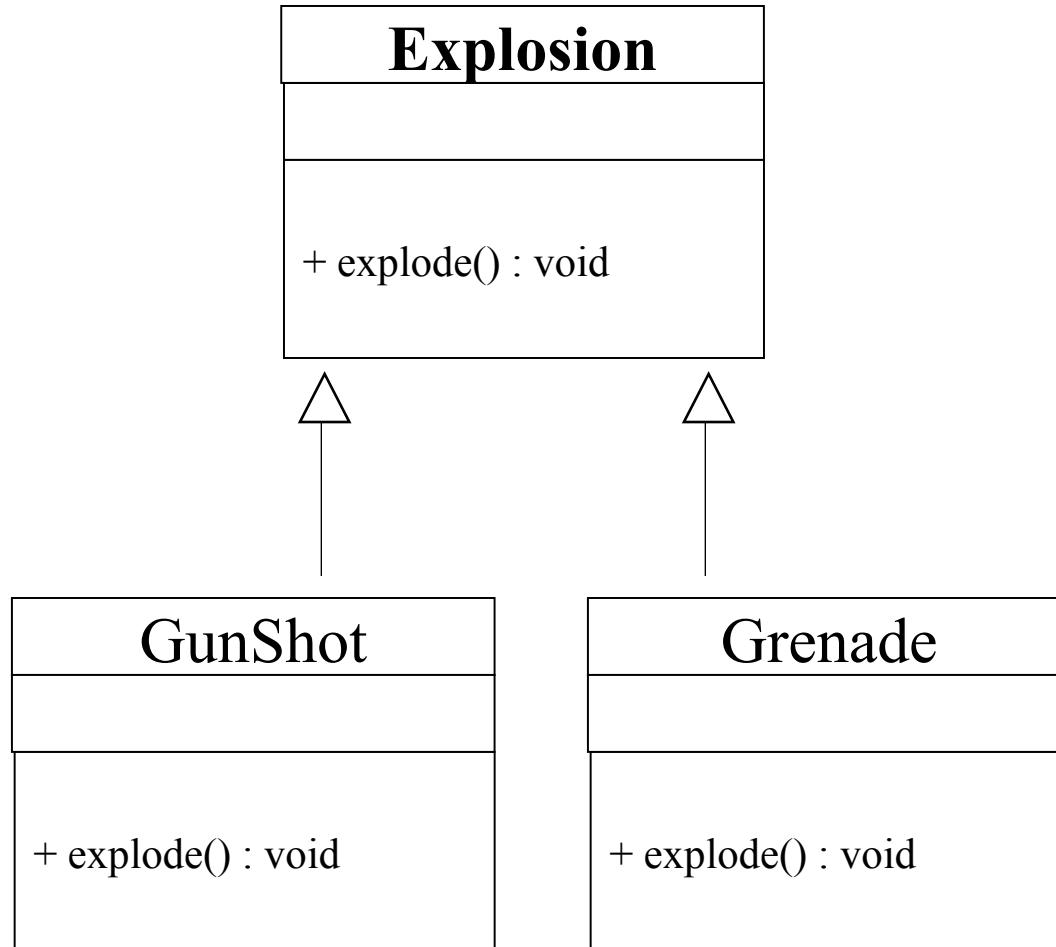
2. Design **concrete derived target subclasses**

- Example: **GunShot, Grenade**

3. Design *one factory class*

- Factory class instantiates **ALL** derived target subclasses based on supplied criteria *AND SWITCHING LOGIC*

Explosion Class Diagram



Simple Factory Pattern

- NOT mentioned in GoF
- Violates S.O.L.I.D. Design Principles
 - Must be changed for new derived classes
violating Open/Closed principle
 - Sole Responsibility principle
- Switching logic for criteria Complicates unit testing

Simple Factory Pattern Usage

1. Instantiate Simple Factory class
2. Use “get object” method, supplying criteria for specific target
3. Method returns instantiated target object after switching logic to determine which concrete derived subclass meets criteria

Simple Factory Pattern

```
public class SimpleExplosionFactory {  
    enum ExplosionCriteria {  
        GUNSHOT,  
        GRENADE  
    }  
    public Explosion getObject( ExplosionCriteria criteria ) {  
        if (criteria == GUNSHOT) {  
            return new Gunshot();  
        } else {  
            return new Grenade();  
        }  
    }  
}
```

Usage Simple Factory Pattern

```
public static void demo() {  
    // Simple Factory: one factory creates all derived objects  
    SimpleExplosionFactory factory =  
        new SimpleExplosionFactory();  
  
    List<Explosion> explosions = new ArrayList<>();  
  
    explosions.add(factory.getObject(GUNSHOT ));  
    explosions.add(factory.getObject(GRENADE ));  
}
```

Updated Simple Factory Pattern

```
public static void demo() {  
    // Simple Factory: one factory creates all derived objects  
    SimpleExplosionFactory factory =  
        new SimpleExplosionFactory();  
  
    List<AbstractExplosion> explosions = new ArrayList<>();  
  
    explosions.add(factory.getObject(GUNSHOT ));  
    explosions.add(factory.getObject(GRENADE ));  
    explosions.add(factory.getObject(ABOMB));  
}
```

Factory Pattern Benefits

- Best for large projects with many developers
- Provides Loose coupling
 - Eliminates widespread hard coding of object instantiation
 - Flexibility: Easy to change class implementation
 - Robust: Supports Best Practices
 - Code depends on Interface or Abstract class
 - Less dependence on concrete classes
 - Greater flexibility than constructor overloading
 - Business case to limit instantiations

Factory Anti-Pattern Detriments

- Anti-Pattern: Using patterns just to use patterns
 - Usually a not a good fit and only adds complexity
- Skip Factory Pattern
 - Simple small project won't benefit much
 - Small group work little benefit
 - No dependencies
 - Few changes
 - No Business logic

Target Classes: example 1

- AbstractExplosion Super Class
 1. Gunshot concrete target subclass
 2. Grenade concrete target subclass
 3. ABomb concrete target subclass

GoF Factory Method Pattern

- Factory Method Pattern is introduced in GoF Design Patterns Book:
 - **Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses"**
- Creational Design Pattern
 - Abstraction: Functionality Hiding
 - Abstracts Class Constructor

Factory Method Design Pattern

- Anatomy of the Factory Method Design Pattern
 1. Who (are): Constituent Components (parts)
 2. What (happens): Operational
 3. Where (useful): Scenarios
 4. Why (design) Rationale
 5. When (used): Benefits

Factory Method Design Pattern

- Who:

Constituent components, roles or parts participating in the deployment and use of the design pattern

1. Product API

- API for all target objects created by factory

2. Concrete Product implements Product API

3. Factory API used by Application (Creator)

4. Concrete Factory implements Factory API to create a Concrete Product (target object)

Factory Method Design Pattern

- What:

A brief description of what happens with the use of the design pattern

 - 1.Application uses Concrete Factory via Factory API
 - 2.Concrete Factory returns an instance of the Concrete Product (targeted object) conforming to Product API

Factory Method Design Pattern

- Where:
A list of suitable scenarios for the use of the design pattern
 - In a Loosely coupled system where (1 or more of the following):
 1. Class wants to delegate object creation to subclass
 2. Need to Localize knowledge of creation process including any helper subclasses

Factory Method Design Pattern

- Why:

A brief description of rationale behind the design of the design pattern

1. APIs define and use derived objects in an abstract loosely coupled relationship

- Factory method abstracts the creation process to maintain the loosely coupled relationship

Factory Method Design Pattern

- When:

A brief description of the benefits obtained when using the design pattern

 1. Abstraction: Factory eliminates creation of objects directly providing flexibility
 2. Supports loose coupling and S.O.L.I.D. design principles because the derived class used to create the target object isn't explicitly used by application code

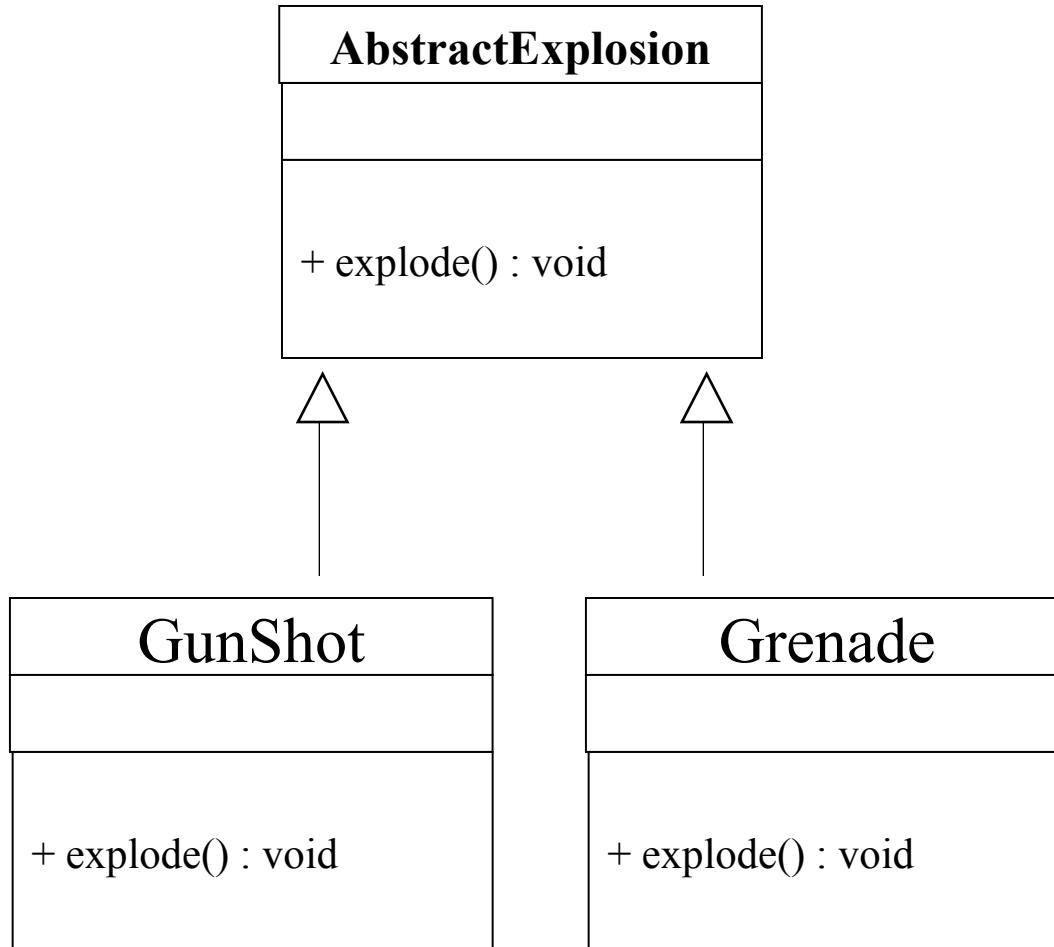
GoF Factory Pattern Method

- Employs Inheritance
- Abstract class or interface as API for target subclasses
 - Just like Simple Factory Pattern
- Abstract class as API for Factory classes
 - Creates instances of a concrete derived Factory subclasses
 - Each Concrete derived factory subclass creates an instance of a concrete derived target subclass
 - Return target object to caller

Factory Pattern Method Components

1. Design **target Abstract super class**
2. Design **concrete derived target subclasses**
3. Design *factory Abstract super class*
4. Design *concrete derived factory subclasses*
 1. Each concrete derived factory class specializes in instantiating one concrete derived target class
 2. No SWITCHING LOGIC simplifies unit testing.

AbstractExplosion Class Diagram



AbstractExplosion

```
public abstract class AbstractExplosion {  
    public abstract void explode();  
}
```

- API for each derived object returned by a factory

GunShot

```
public abstract class GunShot extends  
AbstractExplosion {  
    @Override  
    public abstract void explode();  
}
```

- Each derived object implements AbstractExplosion API

Grenade

```
public abstract class Grenade extends  
AbstractExplosion {  
    @Override  
    public abstract void explode();  
}
```

- Each derived object implements AbstractExplosion API

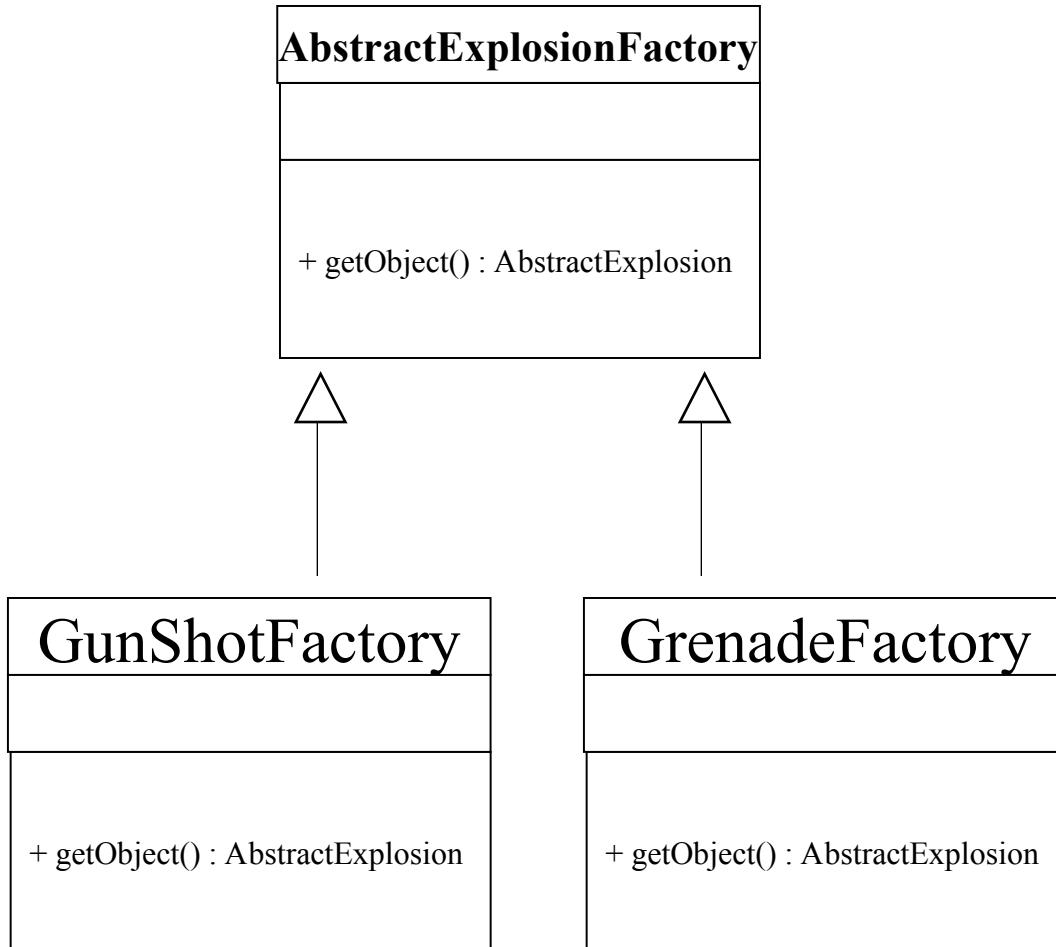
AbstractExplosionFactory

```
public abstract class AbstractExplosionFactory {  
    public abstract AbstractExplosion getObject();  
}
```

- API for used by each factory to create a derived explosion object

AbstractExplosionFactory

Class Diagram



Gunshot Factory derived subClass

```
public class GunshotFactory extends  
AbstractExplosionFactory {  
    public AbstractExplosion getObject() {  
        return new Gunshot();  
    }  
}
```

Grenade Factory derived subClass

```
public class GrenadeFactory extends  
AbstractExplosionFactory {  
    public AbstractExplosion getObject() {  
        return new Grenade();  
    }  
}
```

ABomb Factory derived subClass

```
public class ABombFactory extends  
AbstractExplosionFactory {  
    public AbstractExplosion getObject() {  
        return new ABomb();  
    }  
}
```

Usage Factory Pattern

```
public static void demo() {  
    // Factory Pattern: each factory specializes in one derived  
    objects  
    AbstractExplosionFactory factory1 = new GunshotFactory();  
    AbstractExplosionFactory factory2 = new GrenadeFactory();  
    AbstractExplosionFactory factory3 = new ABombFactory();  
  
    List<AbstractExplosion> explosions = new ArrayList<>();  
    explosions.add(factory1.getObject());  
    explosions.add(factory2.getObject());  
    explosions.add(factory3.getObject());  
}
```

Factory Design Patterns Summary

- Simple Factory Pattern
 - Single Factory uses supplied criteria and switching logic to create one of several specific instantiated target subclasses
- GoF Factory Pattern Method
 - Several Factories EACH of which can create A SINGLE specific instantiated target subclass.

CSYE 6200

Concepts of Object Oriented Design

Java 8 Lambda, Streams

Daniel Peters

d.peters@neu.edu

-
- Lecture
 1. Functional Programming
 2. Lambda
 3. Functional Interface
 4. Generic
 5. Streams
 6. Predicate
-

Functional Programming

- Imperative Programming Style
 - Low Level
 - Step by step instruction
 - Define “What” by implementing “How”
- Declarative Programming Style
 - High Level
 - Implementation is covered
 - Declare ‘What’ without so much ‘How’

Find Dan: Imperative

```
final List<String> names = Arrays.asList("jim",
"sue", "dan", "len", "zac");
boolean found = false;
for (String name : names) {
    if (name.equals("dan")) {
        found = true;
        break; // exit loop
    }
}
System.out.println("You found dan! " + found);
```

OUTPUT: *You found dan!* true

Find Dan: Declarative

```
final List<String> names = Arrays.asList("jim",
"sue", "dan", "len", "zac");
```

```
System.out.println("You found dan! "
+ names.stream()
.filter(s -> s == "dan")
.forEach(System.out::print);
);
```

OUTPUT: *You found dan!* true

Find Number 7: Declarative

```
Integer[] a = {0,1,2,3,4,5,6,7,8,9}; // auto-boxing
```

```
List<Integer> mutableNumbers = new  
ArrayList<Integer>(Arrays.asList(a));
```

```
System.out.print("\n Filter 7 from numbers 0123456789: ");  
mutableNumbers.stream()  
    .filter(n -> n == 7)  
    .forEach(System.out::print);  
System.out.println();
```

OUTPUT: Filter 7 from numbers 0123456789: 7

Lambda

- A.K.A. Closure
- Another Java 8 new feature (like Stream API)
- Borrowed from other languages like: Lisp, Clojure, Erlang, Ruby
- Also exists in other JVM languages like Scala and Groovy
- Allows for Functional Programming in Java

Functionality passable as Data

- Anonymous Class <https://www.baeldung.com/java-anonymous-classes>
 - Usually, an attempt to pass functionality as an argument to another method
 - Unclear cumbersome syntax
- Lambda can similarly treat code as data
 - Concise and compact
 - Anonymous Methods

Lambda

- SYNTAX

`() -> {}`

- ‘`0`’ list of formal supplied parameters
 - `param1, param2, ...`
 - Can omit type
 - Can omit parenthesis for single parameter
- ‘`->`’ Arrow token

Lambda

- SYNTAX

$() \rightarrow \{ \}$

– ‘{}’ body

1. Single expression
2. Statement block
3. Curley braces may be omitted
4. May omit ‘return’ keyword

Example Lambda Parameter List

- Example Parameter list:
 1. ()
 2. p
 3. (int x, int y)
 4. (x,y)
- Parenthesis are optional for a single parameter; required for a multi parameter list
- Optional Parameter type declaration

Example Lambda Expression

1. Example Body Expression:

1. $4.0 == \text{student.getGpa}()$
2. $18 \geq \text{person.getAge}()$
 $\quad \&\& \text{person.getAge}() \leq 35$

- Java runtime will evaluate expression and return its value
- Braces optional for single statement expression

Functional Interface Example

java.util.Function

- Functional Method: Single abstract method
called Annotated: `@FunctionalInterface`
- Target Types
 - Lambda expression

```
Arrays.sort(rosterAsArray,
            (a, b) -> Person.compareByAge(a, b));
```
 - Method reference

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

Example Lambda

- Use Lambda as in-line implementation of a Functional Interface

```
class LambdaDemo {  
    @FunctionalInterface  
    public interface GreatDivide {  
        int divide(int t1, int t2);  
    }  
    public void simpleLambda() {  
        GreatDivide intDivide = (int x, int y) -> x / y;  
        System.out.println(intDivide.divide(21, 3)); // 7 to stdout  
    }  
}
```

Example Lambda

- Use inner class rAnonymous as implementation of the Java Runnable Interface

```
void runnableAnonymous() {  
    Runnable rAnonymous = new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Run rAnonymous, run!");  
        }  
    };  
    Thread t = new Thread(rAnonymous);  
    t.start();      // begin execution on new Thread  
}
```

Example Lambda

- Use Anonymous inner class as implementation of the Java Runnable Interface

```
void runnableAnonymous() {  
    Thread t = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Run anonymous, run!");  
        }  
    }  
};  
t.start();  
}
```

Example Lambda

- Use Lambda as in-line implementation of the Java Runnable Interface

```
void runnableLambda() {  
    Runnable r = () -> System.out.println("run thread in background...");  
    Thread t = new Thread(r);  
    t.start();  
}
```

Example Lambda

- Use Lambda as in-line implementation of the Java Runnable Interface

```
void runnableLambda() {  
    Thread t = new Thread(  
        () -> System.out.println("run thread in background...")  
    );  
    t.start();  
}
```

Example Lambda

- Use Lambda as in-line implementation of the Java Runnable Interface

```
void runnableLambda() {  
    Runnable r = () -> System.out.println("run");  
    Thread t1 = new Thread(r);  
    t1.start();  
    Thread t2 = new Thread(  
        () -> System.out.println("run") );  
    t2.start();  
}
```

String Collection Show method

```
void showList(String title, List<String> list)
{
    System.out.println(l.size() + title);
    for (String n : list) {
        System.out.print(n + ", ");
    }
    System.out.println();
}
```

Generic Collection Show method

```
<E> void showList(String title, List<E> l) {  
    System.out.println(l.size() + title);  
    int i = 1;  
    for (E n : l) {  
        System.out.print(i + ". " + n + " ");  
        i++;  
    }  
    System.out.println();  
}
```

Example Lambda

- Use Lambda as in-line implementation of the Show Generic Functional Interface
- Creates a customized method for the SomeCommodity class

```
@FunctionalInterface
```

```
public interface Show<T> {  
    void show(T t);  
}
```

Example Lambda

```
List<SomeCommodity> shoppingList = new ArrayList<>();  
shoppingList.add(new SomeCommodity("iPhone", 399));  
shoppingList.add(new SomeCommodity("iPad", 599));  
shoppingList.add(new SomeCommodity("macBook", 1599));  
  
// Implement the Show Functional interface with a Lambda  
Show<SomeCommodity> showPrice = o ->  
System.out.println(o.getName() + " only $" + o.getPrice());  
  
for (SomeCommodity item : shoppingList) {  
    showPrice.show(item);  
}
```

Example Stream API

```
public void simpleStream() {  
    List<Integer> list = Arrays.asList(5,2,4,1,3);  
    list.forEach(n -> System.out.print(n + " "));  
    System.out.println("reduce to sorted odd subset");  
    list.stream()  
        .filter(n -> n % 2 == 1)          // odd ONLY  
        .sorted()           // ascending  
        .map(n -> 100*n)      // scale by 10  
        .forEach(n -> System.out.print(n + ", "));  // output  
    System.out.println();  
}
```

OUTPUT: 100, 300, 500,

Predicate

java.util.function

Interface Predicate<T>

public interface Predicate<T>

- Represents a function which returns a boolean value (i.e. a boolean-valued function, a predicate) that accepts a single argument.

- Method

boolean Test(T t)

Predicate Example

```
public void simplePredicate () {  
    List<Integer> ints = Arrays.asList(1,2,3,4,5,6,7,8,9);  
    Predicate<Integer> over5Predicate = n -> { return n >  
5; };  
    for (Integer n : ints) {  
        if (over5Predicate.test(n)) {  
            System.out.print(n + " ** ");  
        } else {  
            System.out.print(n + " ");  
        }  
    }  
}
```

OUTPUT: 1 2 3 4 5 6 ****** 7 ****** 8 ****** 9 ******

Using a Predicate for Stream Filter

- Given the Collection ‘states’

List<String> states =

```
Arrays.asList("ma","ny","ct","vt","ri","nh","nv","nc","nd","wa","wv",
"ut","ca","az","al","ak","ok","pa","me","ms","il","id","mn","wy","mt",
"wi","ia","ar","hi","sd","sc","md","nj","de","ga","fl","mi","oh","in","o
r","ky","tn","va","mo","ks","co","la","tx","nm","ne");
```

- And the Predicate ,‘uStates’

Predicate<String> *uStates*= s -> { return s.startsWith("u"); }

;

Example for Stream Filter

- The following Stream Processing sourced by the ‘states’ String Collection filters using a lambda:

```
states.stream()  
    .filter(s -> s.startsWith("u"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(s -> System.out.print(s + ", "));
```

- Produces the OUTPUT:

UT,

Using a Predicate for Stream Filter

- The following Stream Processing sourced by the ‘states’ String Collection filters using the *uStates* predicate:

```
states.stream()  
    .filter(uStates)  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(s -> System.out.print(s + ", "));
```

- Produces the OUTPUT:

UT,

Imperative: Total Price

```
final List<Double> prices  
=Arrays.asList(5.0,10.0,15.0,20.0);  
double totalOfDiscountedPrices = 0.0;  
  
for (double price : prices) {  
    totalOfDiscountedPrices += price * 0.9;      // 10 %  
    discount  
}  
System.out.println("Total: $" + totalOfDiscountedPrices);
```

OUTPUT: *Total: \$45.0*

Declarative: Total Price

```
final List<Double> prices  
=Arrays.asList(5.0,10.0,15.0,20.0);  
  
final Double totalOfDiscountedPrices =  
    prices  
    .stream()  
    .mapToDouble((Double price) -> price * 0.9)  
    .sum();
```

```
System.out.println("Total: $" + totalOfDiscountedPrices);
```

OUTPUT: *Total: \$45.0*

Method Reference

- Static method Reference

SomeClass::aStaticMethod

- Object instance method Reference

myObject::anInstanceMethod

- Constructor method Reference

SomeClass::new

Method Reference

*“Method references ... are compact, easy-to-read **lambda** expressions for methods that already have a name.”*

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

Method Reference

- Lambda is used to create an anonymous method that can be passed as data, i.e. as formal parameter in sort method.
- Used Method Reference in place of lambda when *only a reference to a method* is required
- Refers to Method by name

Static Method Reference

```
List<String> list
```

```
= Arrays.asList("Jen", "Zac", "Dan");
```

```
list.forEach(System.out::print);
```

```
System.out.println(list.size() + " elements in  
above list.");
```

Object Instance Method Reference

```
class ComparisonProvider {  
    public int compareByName(Person a, Person b) {  
        return a.getName().compareTo(b.getName());  
    }  
    public int compareByAge(Person a, Person b) {  
        return a.getBirthday().compareTo(b.getBirthday());  
    }  
}  
ComparisonProvider myComparisonProvider = new  
ComparisonProvider();  
Arrays.sort(rosterAsArray,  
myComparisonProvider::compareByName);
```

Online Information: Lambda Links

- Oracle:

<https://community.oracle.com/docs/DOC-1003597>

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax>

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>

Online Information: Method Links

- Method Reference

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

Online Information: Streams Links

- Streams

<http://www.oracle.com/technetwork/articles/java/mal4-java-se-8-streams-2177646.html>

CSYE 6200

Concepts of Object-Oriented Design

Review

Daniel Peters

d.peters@neu.edu

-
- Lecture
 1. UML
 2. API, Object Oriented and S.O.L.I.D. Principles
 3. Class, Object, Interface, Inner Classes
 4. Data types, parameter passing
 5. Generics, Collections, Sort, CSV parsing
 6. Design Patterns
 7. Lambda
-

UML

- Picture Language
- Class, Object Diagrams
- Relationships
 - Has-A
 - Association:
 - Aggregation: Weak: linked independent
 - Composition: Strong: dependent
 - Is-A
 - Generalization
 - Inheritance

Object Oriented Design

- Object Oriented (OO) Design Principles
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism

Abstraction

- Data Hiding
 - Access modifiers: public, private, protected, (default)

<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

- Functionality Hiding
 - Application Programming Interface (API)
 - Superclass
 - Concrete class, Abstract class, Interface
 - API can specify *WHAT* without implementing *HOW*

Encapsulation

- Data and Methods co-located in same class
 - Private data
 - Public methods which operate on private data

Inheritance

- Java supports ONLY Single Inheritance
 - Keyword: **extends**
 - *Single* Concrete class OR *single* Abstract class
 - Is-A relationship
 - Derived child class type IS-A Superclass parent type, e.g. GunShot extends Explosion therefore GunShot Is-A Explosion
 - Keyword: **implements**
 - *Multiple* Interfaces

Polymorphism

- Static Polymorphism
 - Overload methods in the *same* class
 - Compile-time polymorphism
- Dynamic Polymorphism
 - Derived *subclass* Override method of a *super* class
 - Run-time polymorphism

Static Polymorphism

- Requirements for Static Polymorphism
 - 1. Method in same class with identical name
 - 2. But method differ
 - Argument (passed parameter) type
 - Number of Arguments

Dynamic Polymorphism

- Requirements: Dynamic Polymorphism
 1. *Inheritance*: super (parent) and subclass (derived, child) classes
 2. *Non-Final*: super class must specify method that is NOT final
 3. *Override*: subclass must override non-final method in super class
 4. Super class type reference variable: must use *variable of type super class to reference* derived object to demonstrate subclass implementation when calling super class method (dynamic polymorphism is subclass method overriding super class non-final method at run-time).
-

S.O.L.I.D. Design Principles

- Sole responsibility principle
- Open-Closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
 - Based on OO principles
 - Abstraction, Inheritance, Polymorphism
 - Provides for Loose Coupling through API
 - Requires the three O.L.D. of S.O.L.I.D. principles

Class

- A specification or blueprint of all members
 - Data members of class
 - Method members of class
- Details
 - Encapsulation implemented (co-located) with:
 1. Private data
 2. Public methods which operate on private data

Class

- Details
 - Static:
 - ALL objects share same static data; one copy
 - Class has static method that does not require object instantiation
 - Program scope: always available

Class

- Details
 - Object Instantiation from Class specification
 - Non-static object instance data:
 - Each object has it's own data copy
 - Object method requires object instantiation
 - Does NOT exist until object instantiation
 - » Scope limited by usage
 - » Automatic garbage collection

Object

- Instantiation (creation):
 - Must use keyword ‘new’ followed by class constructor

```
Person p = new Person();
```

```
System.out.println(p.info());
```

```
Person p = new Person(“Jim”, “Doe”, 19);
```

```
System.out.println(p.info());
```

- Always allocated on heap
 - Always a reference type
 - Two distinct memory allocations:
 1. Heap allocated object
 2. Stack allocated reference to object

Interface

- Comparable
 - Implement to give class objects default or natural ordering for sort (without need for a Comparator)
- Comparator
 - Implement to explicitly specify a sort order
- Runnable
 - Implement to specify a run() method with code to execute on a new thread

Info

- Exam will have an on-line portion and a programming portion. You may use only your IDE.
- Complete all work before time expires.
 - Late submissions receive a severe grade deduction
- On-Line exam means **video must be on**
- Taking exam in **violation of any** requirements incurs a **grade of zero (0)**.

Excluded Topics

- You will not be required to write code to perform file Input Output
 - No need to write code to read or write files
 - BUT you must understand String CSV parsing

Included Topics

- Topics *may* include (but are **not limited to**) the following:
 - Class, Object, Abstract Class, Interface
 - Object Oriented Design principles
 - S.O.L.I.D. principles
 - Data types: Primitive, Reference, enum, parameter passing
 - Arrays, Collections (sequential, associative), Sort
 - UML
 - Design Patterns: Singleton, Simple Factory and GoF Factory
 - String CSV parsing
 - Exceptions
 - API, Loose Coupling