Building Java Programs

Inheritance and Polymorphism
Input and output streams

- **stream**: an abstraction of a source or target of data
  - 8-bit bytes flow to (output) and from (input) streams

- can represent many data sources:
  - files on hard disk
  - another computer on network
  - web page
  - input device (keyboard, mouse, etc.)

- represented by `java.io` classes
  - `InputStream`
  - `OutputStream`
Recall: inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/reuse code between two or more classes
- **superclass**: Parent class being extended.
- **subclass**: Child class that inherits behavior from superclass.
  - gets a copy of every field and method from superclass
- **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.
Streams and inheritance

- input streams extend common superclass `InputStream`;
- output streams extend common superclass `OutputStream`
- guarantees that all sources of data have the same methods
- provides minimal ability to read/write one byte at a time
Inheritance syntax

public class name extends superclass {

public class Lawyer extends Employee {

... 
}

• **override**: To replace a superclass's method by writing a new version of that method in a subclass.

    public class Lawyer extends Employee {
        // overrides getSalary method in Employee class;
        // give Lawyers a $5K raise
        public double getSalary() {
            return 55000.00;
        }
    }
}
super keyword

- Subclasses can call inherited behavior with `super`

```java
super.method(parameters)
super(parameters);
```

```java
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years); // calls Employee constructor
    }

    // give Lawyers a $5K raise
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

- Lawyers now always make $5K more than Employees.
I/O and exceptions

- **exception**: An object representing an error.
  - **checked exception**: One that must be handled for the program to compile.

- Many I/O tasks throw exceptions.
  - Why?

- When you perform I/O, you must either:
  - also **throw** that exception yourself
  - **catch** (handle) the exception
Throwing an exception

```java
public type name(params) throws type {
    // Throws clause: Keywords on a method's header that state that it may generate an exception.
    // Example:
    public void processFile(String filename)
    throws FileNotFoundException {
        "I hereby announce that this method might throw an exception, and I accept the consequences if it happens."
Catching an exception

```java
try {
    statement(s);
} catch (type name) {
    code to handle the exception
}
```

- The `try` code executes. If the given exception occurs, the `try` block stops running; it jumps to the `catch` block and runs that.

```java
try {
    Scanner in = new Scanner(new File(filename));
    System.out.println(input.nextLine());
} catch (FileNotFoundException e) {
    System.out.println("File was not found.");
}
```
Exception inheritance

- Exceptions extend from a common superclass `Exception`

![Exception Inheritance Diagram]
Dealing with an exception

- All exception objects have these methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String getMessage()</td>
<td>text describing the error</td>
</tr>
<tr>
<td>public String toString()</td>
<td>a stack trace of the line numbers where error occurred</td>
</tr>
<tr>
<td>getCause(), getStackTrace(), printStackTrace()</td>
<td>other methods</td>
</tr>
</tbody>
</table>

- Some reasonable ways to handle an exception:
  - try again; re-prompt user; print a nice error message; quit the program; do nothing (!)
Inheritance and exceptions

- You can catch a general exception to handle any subclass:

```java
try {
    Scanner input = new Scanner(new File("foo"));
    System.out.println(input.nextLine());
} catch (Exception e) {
    System.out.println("File was not found.");
}
```

- Similarly, you can state that a method throws any exception:

```java
public void foo() throws Exception { ... }
```

- Are there any disadvantages of doing so?
The class Object

- The class **Object** forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of **Object**

- The **Object** class defines several methods that become part of every class you write. For example:
  - `public String toString()`
    Returns a text representation of the object, usually so that it can be printed.
## Object methods

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected Object <code>clone()</code></td>
<td>creates a copy of the object</td>
</tr>
<tr>
<td>public boolean <code>equals(Object o)</code></td>
<td>returns whether two objects have the same state</td>
</tr>
<tr>
<td>protected void <code>finalize()</code></td>
<td>used for garbage collection</td>
</tr>
<tr>
<td>public Class&lt;?&gt; <code>getClass()</code></td>
<td>info about the object's type</td>
</tr>
<tr>
<td>public int <code>hashCode()</code></td>
<td>a code suitable for putting this object into a hash collection</td>
</tr>
<tr>
<td>public String <code>toString()</code></td>
<td>text representation of object</td>
</tr>
<tr>
<td>public void <code>notify()</code></td>
<td>methods related to concurrency and locking (take a data structures course!)</td>
</tr>
<tr>
<td>public void <code>notifyAll()</code></td>
<td></td>
</tr>
<tr>
<td>public void <code>wait()</code></td>
<td></td>
</tr>
<tr>
<td>public void <code>wait(...)</code></td>
<td></td>
</tr>
</tbody>
</table>
Using the Object class

- You can store any object in a variable of type `Object`.
  ```java
  Object o1 = new Point(5, -3);
  Object o2 = "hello there";
  ```

- You can write methods that accept an `Object` parameter.
  ```java
  public void checkNotNull(Object o) {
    if (o != null) {
      throw new IllegalArgumentException();
    }
  }
  ```

- You can make arrays or collections of `Objects`.
  ```java
  Object[] a = new Object[5];
  a[0] = "hello";
  a[1] = new Random();
  List<Object> list = new ArrayList<Object>();
  ```
Recall: comparing objects

- The `==` operator does not work well with objects.
  - It compares references, not objects' state.
  - It produces `true` only when you compare an object to itself.

```java
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true

// p1.equals(p2)?
// p2.equals(p3)?
```
Default equals method

- The `Object` class's `equals` implementation is very simple:
  ```java
  public boolean equals(Object o) {
    return this == o;
  }
  ```

- However:
  - When we have used `equals` with various objects, it didn't behave like `==`. Why not? `if (str1.equals(str2)) { ...`
  - The [Java API documentation for equals](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)) is elaborate. Why?
Implementing equals

```java
public boolean equals(Object name) {
    statement(s) that return a boolean value;
}
```

- The parameter to `equals` must be of type `Object`.
- Having an `Object` parameter means *any* object can be passed.
  - If we don't know what type it is, how can we compare it?
Casting references

Object o1 = new Point(5, -3);
Object o2 = "hello there";

((Point) o1).translate(6, 2); // ok
int len = ((String) o2).length(); // ok
Point p = (Point) o1;
int x = p.getX(); // ok

- Casting references is different than casting primitives.
- Really casting an Object reference into a Point reference.
- Doesn't actually change the object that is referred to.
- Tells the compiler to assume that o1 refers to a Point object.
The `instanceof` keyword

```java
if (variable instanceof type) {
    statement(s);
}
```

- Asks if a variable refers to an object of a given type.
- Used as a boolean test.

```java
String s = "hello";
Point p = new Point();
```

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s instanceof Point</td>
<td>false</td>
</tr>
<tr>
<td>s instanceof String</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof Point</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>p instanceof Object</td>
<td>true</td>
</tr>
<tr>
<td>s instanceof Object</td>
<td>true</td>
</tr>
<tr>
<td>null instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>null instanceof Object</td>
<td>false</td>
</tr>
</tbody>
</table>
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
More about equals

- Equality is expected to be reflexive, symmetric, and transitive:
  
  \[
  \begin{align*}
  &a.\text{equals}(a) \text{ is true for every object } a \\
  &a.\text{equals}(b) \iff b.\text{equals}(a) \\
  &(a.\text{equals}(b) \land b.\text{equals}(c)) \iff a.\text{equals}(c)
  \end{align*}
  \]

- No non-null object is equal to null:
  
  \[
  a.\text{equals}(null) \text{ is false for every object } a
  \]

- Two sets are equal if they contain the same elements:
  
  ```java
  Set<String> set1 = new HashSet<String>();
  Set<String> set2 = new TreeSet<String>();
  for (String s : "hi how are you".split(" ")) {
    set1.add(s);
    set2.add(s);
  }
  System.out.println(set1.equals(set2)); // true
  ```
Polymorphism
Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

- A variable or parameter of type $T$ can refer to any subclass of $T$.

  ```java
  Employee ed = new Lawyer();
  Object otto = new Secretary();
  ```

  - When a method is called on `ed`, it behaves as a `Lawyer`.
  - You can call any `Employee` methods on `ed`.
  - You can call any `Object` methods on `otto`.
    - You can *not* call any `Lawyer-only` methods on `ed` (e.g. `sue`).
    - You can *not* call any `Employee` methods on `otto` (e.g. `getHours`).

  ...
Polymorphism examples

- You can use the object's extra functionality by casting.

```java
Employee ed = new Lawyer();
ed.getVacationDays(); // ok
ed.sue(); // compiler error
((Lawyer) ed).sue(); // ok
```

- You can't cast an object into something that it is not.

```java
Object otto = new Secretary();
System.out.println(otto.toString()); // ok
otto.getVacationDays(); // compiler error
((Employee) otto).getVacationDays(); // ok
((Lawyer) otto).sue(); // runtime error
```
"Polymorphism mystery"

- Figure out the output from all methods of these classes:

```java
public class Snow {
    public void method2() {
        System.out.println("Snow 2");
    }
    public void method3() {
        System.out.println("Snow 3");
    }
}

public class Rain extends Snow {
    public void method1() {
        System.out.println("Rain 1");
    }
    public void method2() {
        System.out.println("Rain 2");
    }
}
```
public class **Sleet** extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        super.method2();
        method3();
    }
    public void method3() {
        System.out.println("Sleet 3");
    }
}

public class **Fog** extends Sleet {
    public void method1() {
        System.out.println("Fog 1");
    }
    public void method3() {
        System.out.println("Fog 3");
    }
}
Technique 1: diagram

- Diagram the classes from top (superclass) to bottom.
# Technique 2: table

<table>
<thead>
<tr>
<th>method</th>
<th>Snow</th>
<th>Rain</th>
<th>Sleet</th>
<th>Fog</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td></td>
<td>Rain 1</td>
<td></td>
<td>Fog 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method2</td>
<td>Snow 2</td>
<td>Rain 2</td>
<td>Sleet 2</td>
<td>Sleet 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Snow 2</td>
<td>Snow 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>method3()</td>
</tr>
<tr>
<td>method3</td>
<td>Snow 3</td>
<td>Snow 3</td>
<td>Sleet 3</td>
<td>Fog 3</td>
</tr>
</tbody>
</table>

*Italic* - inherited behavior  
*Bold* - dynamic method call
Mystery problem, no cast

```java
Snow var3 = new Rain();
var3.method2(); // What's the output?
```

- If the problem does *not* have any casting, then:
  1. Look at the *variable*'s type.
     If that type does not have the method: ERROR.
  2. Execute the method, behaving like the *object*'s type.
     (The variable type no longer matters in this step.)
Example 1

- What is the output of the following call?

  ```java
  Snow var1 = new Sleet();
  var1.method2();
  ```

- Answer:

  ```text
  Sleet 2
  Snow 2
  Sleet 3
  ```
Example 2

- What is the output of the following call?

```java
Snow var2 = new Rain();
var2.method1();
```

- Answer:

ERROR (because `Snow` does not have a `method1`)

![Diagram showing the hierarchy of classes and methods](image)
Mystery problem with cast

```java
Snow var2 = new Rain();
((Sleet) var2).method2(); // What's the output?
```

- If the problem *does* have a type cast, then:
  1. Look at the *cast* type.
     If that type does not have the method: ERROR.
  2. Make sure the *object's* type is the *cast* type or is a subclass of the cast type. If not: ERROR. (No sideways casts!)
  3. Execute the method, behaving like the *object's* type.
     (The variable / cast types no longer matter in this step.)
Example 3

- What is the output of the following call?

```java
Snow var2 = new Rain();
((Rain) var2).method1();
```

- Answer:

```
Rain 1
```
Example 4

- What is the output of the following call?

```java
Snow var2 = new Rain();
((Sleet) var2).method2();
```

- Answer:

```
ERROR
(because the object's type, Rain, cannot be cast into Sleet)
```