CMPT 225: Data Structures & Programming
– Unit 03 –
Object Oriented Design, Part Two

Dr. Jack Thomas

Simon Fraser University

Spring 2021

Today's Topics

- Abstract Data Types
- Abstract Classes
- Interfaces
- Throwables
- Casting

Addendum to Last Lecture

- If a subclass is instanced as its superclass (i.e. a CowboyGreeting object created as a Greeting), you can't access the overloaded versions of its functions, but the overridden ones will still be overridden (i.e. printing "howdy partner!").
- A function's signature is its name and the parameters it accepts, not the return type.
- Possibility for future lectures: saving some technical questions until the end to be answered next time to preserve the lecture's flow?

Abstract Data Types

- As mentioned, an **Abstract Data Type** (ADT) is the idea behind a data structure, like a stack or a queue, rather than the structure itself.
- A **Concrete Data Type**, therefore, is the actual implementation of the ADT, well-defined in the actual code.
  - It's rare we actually need to use this term.
- ADTs collectively represent the objects that make up object-oriented programming, leaving it to individual languages and programmers to implement them.

What ADTs Provide

- ADTs typically define the types of data they store and the operations they allow – **what** they are, but not **how** they work.
- Common methods include:
  - **Mutators**: How the data inside the object can be manipulated (“setters”).
  - **Accessors**: What data the object makes available to others (“getters”).
  - **Constructors**: Different ways to build the object.
- These represent the object’s **Application Program Interface** (API), or interface.

Using Abstraction in Java

- It's possible to declare a **class** to be **Abstract**, meaning it can only be inherited from, not instanced on its own.
- This allows **Abstract functions** to be declared with only a signature, that all extending subclasses will have to fill in on their own.
- It otherwise works as a normal class (it can have variables and other, completed functions, or not).

```java
abstract class Greeter
{
    String name = "Default";
    public abstract void greet();
}

class NormalGreeter extends Greeter
{

    public void greet()
    {
        System.out.println("Hello, I'm " + name + ".");
    }
}

class CowboyGreeter extends Greeter
{
    public void greet()
    {
        System.out.println("Howdy, I'm " + name + "!");
    }
}
```

```java
NormalGreeter normalGuy = new NormalGreeter();
CowboyGreeter cowboyGuy = new CowboyGreeter();
normalGuy.greet();
cowboyGuy.greet();
```

```
Hello, I'm Default.
Howdy, I'm Default!
```

Image Credit: https://www.amazon.com/Faux-Felt-Wide-Western-Cowboy/dp/B00LWZMO88

Java Interfaces

- **Interfaces** (the Java code term, not the abstract concept) are made up entirely of Abstract function signatures, with no variables or completed functions.
- Classes that **implement** an interface must then fill in these functions.
- Unlike other forms of inheritance in Java, classes can implement multiple interfaces.

```java
interface SmallTalk{
    public void randomChatter();
}

interface AwkwardDeflection{
    public void weatherComment();
}
```

```java
class NormalGreeter extends Greeter implements SmallTalk, AwkwardDeflection
{

    public void greet()
    {
        System.out.println("Hello, I'm " + name + ".");
    }
    public void randomChatter()
    {
        System.out.println("So, how're you holding up?");
    }
    public void weatherComment()
    {
        System.out.println("This is some weather we're having, huh?");
    }
}
```

Brief Aside: Object Orientation in Java vs. in Other Languages

- We are **mixing** some of the **general** ideas of the object-oriented paradigm with the **specific** implementation decisions of Java.
- **Different languages** have found other ways to implement some of these principles.
- For example, **in C++**, classes can **inherit from multiple other classes** – functions simply include from what class they derive as part of their signature.

Errors, Exceptions, and Throwing

- In Java, there are two broad categories of problem – **Errors**, where the code will break, or **Exceptions**, when we want the code to stop for our own reasons.
- Both are objects, subclasses of **Throwable**, because when their conditions are met the program will **Throw** one of them.
- Programmers can define their own conditions for when the code should Throw to prevent undesirable or code-breaking outcomes.

```java
class NumberAdder
{
    int[] myFiveNumbers = {10, 20, 30, 40, 50};
    public void swapANumber(int newNumber, int indexToSwap)
    {
        myFiveNumbers[indexToSwap] = newNumber;
        System.out.println("Okay! Now the number at index " + indexToSwap + " is " + newNumber);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 25 out of bounds for length 5
```

```java
class NumberAdder
{
    int[] myFiveNumbers = {10, 20, 30, 40, 50};
    public void swapANumber(int newNumber, int indexToSwap)
    {
        if (indexToSwap > myFiveNumbers.length) {
            throw new ArrayIndexOutOfBoundsException("The index you chose is too high!");
        }
            myFiveNumbers[indexToSwap] = newNumber;
            System.out.println("Okay! Now the number at index " + indexToSwap + " is " + newNumber);

    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: The index you chose is too high!
```

Try and Catch

- Once a Throwable object is created, we need to use it to handle the problem.
- When we expect an error or exception might occur, we can enclose that piece of code in a **Try** block, and then follow it with a **Catch** block that triggers if something is thrown.
- Think of it as an If/Else block, where the condition is met if a Throwable of the correct type is created.

```
class NumberAdder
{
    int[] myFiveNumbers = {10, 20, 30, 40, 50};
    public void swapANumber(int newNumber, int indexToSwap)
    {
        try {
            if (indexToSwap > myFiveNumbers.length) {
                throw new ArrayIndexOutOfBoundsException("The index you chose is too high!");
            }
            myFiveNumbers[indexToSwap] = newNumber;
            System.out.println("Okay! Now the number at index " + indexToSwap + " is " + newNumber);
        }
        catch (ArrayIndexOutOfBoundsException myException) {
            System.out.println("The index you gave me was " + indexToSwap + ", that's too big!");
        }

    }
}
```

```
The index you gave me was 25, that's too big!

Anyway back to whatever else this program does.
Mostly cowboy stuff for some reason?
```

(Yes, you can have multiple catch blocks for different Throwables)

Casting

- A lot of the techniques that object-oriented programming allow take advantage of how inheritance means a class can belong to multiple other classes.
- **Casting** allows us to change how an object is seen.

```
Greeting cowboyPolymorphic = new cowboyGreeting();
cowboyPolymorphic.sayHello();
cowboyGreeting recastCowboy = (cowboyGreeting)cowboyPolymorphic;
recastCowboy.sayHello( numberOfPartners: 2);
```

Widening and Narrowing Conversions

- If an object is recast as something further up its inheritance chain (i.e. Integer to Number, or CowboyGreeting to Greeting), this is a **Widening Conversion**.
- Some data can be lost this way (say, if the wider type doesn't have some variables that the original narrower object had) but it should not cause errors, as the narrower object was also a valid instance of the wider object.
- The reverse case is a **Narrowing Conversion**, but carries more risk – the compiler doesn't know, for example, if the original object is missing some variables that the new narrower case requires, which can lead to run-time errors.

Recap – End Of Slideshow Exception

- **Abstract Data Types** are the general ideas behind data structures in object-oriented programming.
- **Abstract Classes** can only be extended, not instanced.
- Classes can implement multiple **Interfaces**, but all of their functions are just empty signatures.
- **Throwables** are a type of object for containing and handling errors and exceptions that must be caught.
- **Casting** can change what we see an object as, changing the options available to us.
- All of these tools will help us to implement and use the various data structures and algorithms the rest of the course will cover!