Exploring Inheritance

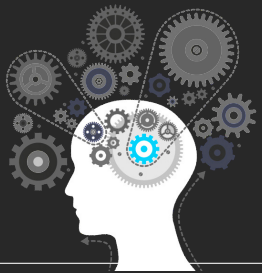
Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will:

- Have a deeper understanding of inheritance
- Know some of the direct applications of inheritance
- Know the pitfalls and limitations of inheritance
- Be able to gain the benefits of inheritance using composition



Exploring Inheritance

CS 2263

Introduction

- Goal of inheritance: **To maximize reuse**
- There are two primary methods for implementing inheritance:
  - Subclassing existing classes
  - Implementing interfaces
- **Note:** Subclassing must be done with care to avoid ending up with an unusable system
  - **Liskov Substituion Principle** provides an elegant test to determine if subclassing is appropriate for a given context

Applying Inheritance

CS 2263

Applications

- Effective inheritance requires a good deal of insight into how a system will evolve

- Towards understanding this we present serval applications of inheritance:
  - Restricting behaviors and properties
  - Abstract superclasses
  - Adding features
  - Hiding features of a superclass
  - Combining structural and type inheritance

Restricting Behaviors and Properties

- Inheritance may be used when we have a class with characteristics which we need to restrict through subclassing
  - Example: `Rectangle` and `Square`

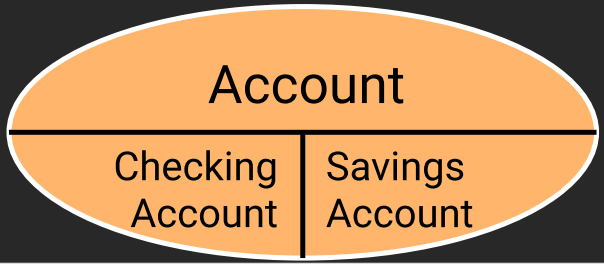
    A `square` adds the restriction where the `length == width`

- Thus, rather than adding functionality we are simply restricting the system

- **We should note, such uses of inheritance are usually NOT justified**

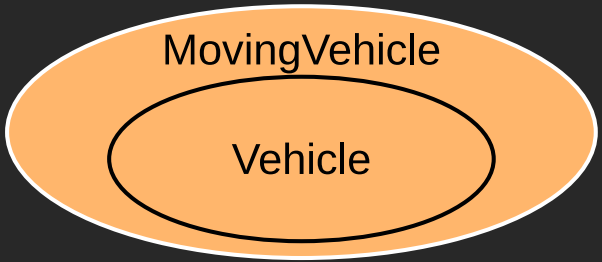
Abstract Superclass

- Sometimes the only purpose of a superclass is to extract out common attributes and methods
  - Thus, maximizing reuse
- No objects of the superclass are allow → Abstract
- We can think of this as a set of **subclasses that partition the universe** of objects in the superclass



Adding Features

- As opposed to restricting behavior, we can also extend a class in order to add new features
- Example: `MovingVehicle` class which extends the `Vehicle` class and adds in the attribute `speed`



Hiding Superclass Features

- We can restrict behavior by suppressing some functionality of the superclass
- This type of inheritance is called **structural inheritance**
  - The superclass provides the structure needed for implementing the subclasses
- **This approach violates the 'is-a' relationship and thus the Liskov Substitution Principle**

Inheritance Limitations

CS 2263

Not always the Best Choice

- Inheritance is not always the best choice for creating new classes
  1. Subclassing may result in **deep hierarchies**
     - decreasing code understandability
  2. If multiple inheritance is not supported by the language
     - subclassing may not even be feasible
  3. It may be necessary to hide features of the superclass
     - otherwise you risk compromising class integrity
     - but, this violates the LSP
  4. Combining inheritance and generics may lead to unintended complications
  5. Derived class's type may not be a true subtype of the superclass's type

Deep Hierarchies

- Each new subclass adds one more to the depth of the hierarchy
- Worse yet, each class in the hierarchy inherits (and thus includes) all of the public and protected fields and methods of all of its ancestor classes

Class Name	Number of Fields	Number of Methods
`java.awt.Component`	186	291
`java.awt.Container`	230	417
`java.swing.JComponent`	376	594
`java.swing.text.JTextComponent`	440	698
`javax.swing.JTextField`	479	729
`javax.swing.JFormattedTextField`	513	757

- This increases both the logical and psychological complexity of the system
  - i.e., it is more difficult to remember all the interactions between methods in the system
  - Instead, we should strive to limit the dept of our hierarchies

Lack of Multiple Inheritance

- In some situations it is desirable to subclass from multiple classes
- Yet, due to an overuse and dangerous practices in C++, multiple inheritance was excluded from Java
- Thus, we must consider other approaches

Changes in the Superclass

- Change is the inevitability in software development
  - Thus, while not necessarily desirable this means we may need to modify a class

- If we change a superclass's functionality
  - Its subclasses immediately benefit from the enhanced functionality
  - But, we can also easily break both the subclasses and the clients of the class
    - If we change a method signature
    - Or we modify the internal behavior upon which clients rely (i.e., side-effects)

Typing Issues: The LSP

- One of the principles centered on the use of inheritance is the **Liskov Substitution Principle**

LSP

Subclasses should be substitutable for their base classes

Barbara Liskov Notes:

If for each object O_1 of type *S* there is an object O_2 of type *T* such that for all programs *P* defined in terms of *T*, the behavior of *P* is unchanged when O_1 is substituted for O_2 then *S* is a subtype of *T*

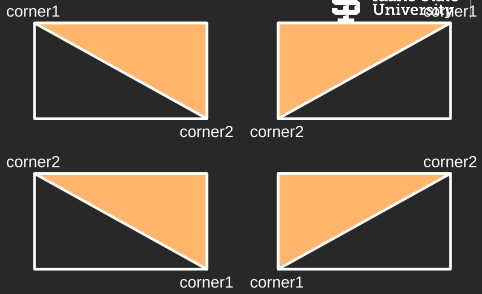
LSP: An Example

- Lets say we have a package which provides the class `SolidRectangle`, which creates a rectangle that is:
  - Solid
  - Axis-Parallel

- Each `SolidRectangle` is then defined by two points
  - The two ends of one of the diagonals
  - We can define the **upper triangle** as that which is formed by the corner above the diagonal and the two defining points

LSP: An Example

- Thus we know the following:
  - When $corner1.y > corner2.y$
    - The third point $= (corner2.x, corner1.y)$
  - Otherwise
    - The third point $= (corner1.x, corner2.y)$



LSP: An Example

Idaho State University | Computer Science

- Now we could conceive of a subclass of `SolidRectangle` which restricts us to a 1x1 rectangle. We will call this class `Pixel`
  - The restriction is: $corner1 = corner2$
  - All setters will be overridden in order to preserve this constraint
- The problems will then arise in the existing clients of `SolidRectangle`
  - If we had a method `getUpperTriangle()` which returns `null` if `corner1.x == corner2.x || corner1.y == corner2.y`
  - Then, if a `Pixel` is provided to a client which relies upon this method, then a `null` would be returned
    - The client would then face a `NullPointerException`
- Even worse, it becomes apparent that `Pixel` is not a true subclass of `Rectangle` as it does not meet the definition set out by LSP in **all** situations.

LSP: An Example

Key Takeaway

- When we extend a class, we must evaluate all behaviors of the class being extended, even in situations where a subset relationship exists in reality between corresponding entities.

Addressing Limitations

- OO approaches to address the limitations are the following two rules of thumb:
  1. Inherit from abstract types rather than concrete classes
  2. Favor composition over inheritance
- In most situations where we consider using inheritance, at least one of these rules (and most likely a combination) will apply.

Type Inheritance

CS 2263

Type Inheritance

- As you may have guessed, we can do more than simply inherit properties, we can also inherit behavior.

- This works as follows:
  - Class `C` provides a function `f` which operates on objects of type `T`
    - that is, it takes a parameter of type `T`: `f(T t) { ... }`
  - Class `D` needs to use this functionality
    - Thus, `D` needs to acquire the type `T`
  - Thus, `D` implements the interface `T`

Simple Example

- In the class `java.util.Collections` there exists the following method:
  - `static <T> sort(List<T> list) {...}`
  - As one may intuit, this method sorts the provided list.
- In order to sort the list, the list must be contain objects which implement the interface `Comparable`
  - Comparable provides only one method: `int compareTo(T object)`
- Let us take a look at an example

The `Clonable` Interface

Idaho State University | Computer Science

- Often we need the ability to copy objects in order to pass objects around without violating the constraints of a system
- Unfortunately, in order to achieve this, we cannot simply copy the reference, or just the values (**a shallow copy**) rather, we need something more.
- Instead, we need to create a **deep copy** which requires knowledge of the internals of the object
- A deep copy, not only copies the primitive values of the object, but also all of the data of the objects which it references as well.

The `Clonable` Interface

- The rule for a deep copy like this is that the object being copied must decide on how it will handle the request to be cloned.
  - This becomes complicated if the object contains many references to other objects
  - Because each of these objects, in turn, must also decide how to handle being cloned

- Java, via `Object`, provides us with a method `clone()` by which this can be implemented.
  - The **default** behavior of `clone()` is to simply provide a **shallow copy**
  - Note, that the base version of `clone()` is marked `protected`
- Java also provides the `CloneNotSupportedException` which can be used to signal that cloning is not supported for a given object.

Attitudes towards cloning

Idaho State University | Computer Science

- A class can take 4 attitudes towards cloning
  1. **Supports Cloning:** The class will need to implement the `Clonable` interface and declare a `public clone()` method which throws **no exceptions**
  2. **Conditionally Supports Cloning:** Class implements `Clonable`, but cannot guarantee its contents can be cloned.
     - Its `clone()` method will throw `CloneNotSupportedException` if some of its contents can't be cloned
  3. **Does Not Publicly Support Cloning, but allows subclasses to support cloning:** Class does not implement `Clonable`, but may override default implementation of `clone()`.
     - Subclasses can then call `super.clone()` if they support cloning
  4. **Does not support cloning:** Class provides a `clone()` method which always throws a `CloneNotSupportedException`

- **Note:** Take care when implementing `clone()`, that you do not have circular references as this will lead to infinite recursion and a `StackOverflowException`

The `Runnable` Interface

- Often, in order to improve performance we need to employ **concurrent sequential processing**
  - Better known as **multithreading**
- Towards this end, Java, provides a class called `Thread`.
- `Thread` allows us to (among other things):
  - Suspend processes
  - Make a process sleep for a period of time
  - Prioritize processes
  - Enable sharing of resources
  - Allow more than one process to execute simultaneously

The `Runnable` Interface

Idaho State University | Computer Science

- Unfortunately, `Thread` is a class, so if we extend it we cannot extend any other class (due to Java's limitation to Single Inheritance)
- We now need to employ another approach to acquire the capabilities of `Thread`
- We do this using one of the following interfaces:
  - `Runnable` - an interface which provides a single method `void run()`
    - `run()` contains the logic/behavior to be executed within a separate thread
    - `run()` takes no parameters (but they can be passed to the constructor of the class)
    - `run()` returns `void`
  - `Callable` - an interface which provides a single method `T call()`
    - `call()` is similar to run, but can return a value

Runnable Example

- Once we have defined our class which implements either `Runnable` or `Callable`
  - Construct a thread and pass it an instance of our class.
  - Using this approach we achieve all the benefits of extending `Thread` via **Object Composition**

```
public class Clock implements Runnable {
  Thread thread = new Thread(this);
  String sound = "tic";

  public void run() {
    try {
      while(true) {
        System.out.println(sound);
        sound = "toc";
        Thread.sleep(1000);
        System.out.println(sound);
        sound = "tic"
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {}
  }
  public Clock() {
    thread.start();
  }

  public static void main(String[] args) {
    new Clock();
  }
}
```

Enhancing the Library Class

Idaho State University | Computer Science

- We will be considering the more sophisticated version of the library system discussed in Chapter 8.

- We have a new feature to add:
  - Our clients wish to expand their collection to include non-print media, including:
    - Books on tape
    - CDs
    - DVDs
    - Periodicals
  - The first 3 will need to keep track of **duration**
  - Recent periodicals cannot be checked out

- For the sake of discussion we will limit ourselves to only periodicals

First Attempt

We first need to ask the following question:

- How do these new requirements change the design of our system?
  - What new classes/methods need to be added?
  - How do the interactions between the existing classes change?

First Attempt

- Changes required:
  - Need to be able to determine which type of item we are dealing with: `Book` or `Periodical`
  - Periodicals do not have an `author`
  - Periodicals less than 3 months old are not `issuable`

- So where do we make the changes?

First Attempt

- We could change the `Book` class
  - Add in a field: `bookType`
    - Tracks which type of book
  - Add in a field: `dateAcquired`
    - Tracks how old the item is
  - Add constants for the type
  - Add constructors specific to the types

Book Changes

Idaho State University | Computer Science

Fields

```
private String title;
private String author;
private String id;
private Member borrowedBy;
private List holds = new LinkedList();
private Calendar dueDate;
private int bookType;
private Calendar dateAcquired;
public static final int BOOK = 1;
public static final int PERIODICAL = 2;
```

Constructors

```
public Book(String title, String author,
            String id) {
  this.title = title;
  this.author = author;
  this.id = id;
  this.bookType = BOOK;
}

public Book(String title, String id) {
  this.title = title;
  this.id = id;
  this.bookType = PERIODICAL;
  this.dateAcquired = new GegorianCalendar();
  this.dateAcquired.setTimeInMillis(
        System.currentTimeMillis());
}
```

Library Changes

- Next we need to update the UI to accommodate the new item type to be added
  - Requires a conditional
  - This requires invoking different constructors

```
public void addBooks() {
  Book result;
  do {
    String title = getToken("Enter title");
    String bookID = getToken("Enter id");
    if (yesOrNo("Is this a book?")) {
      String author = getToken("Enter author");
      result = library.addBook(title, author, bookID);
    } else {
      result = library.addPeriodical(title, bookID);
    }
    if (result != null) {
      System.out.println(result);
    } else {
      System.out.println("Book could not be added");
    }
    if (!yesOrNo("Add more books?")) {
      break;
    }
  } while (true);
}
```

Other Book Methods

- The process for issuing a book is different than the process fo a periodical, thus we need to change `issue()`

```
public Book addBook(String title, String author,
                    String id) {
  Book book = new Book(title, author, id);
  if (catalog.insertBook(book)) {
    return (book);
  }
  return null;
}

// new method added for periodical
public Book addPeriodical(String title,
                          String id) {
  Book book = new Book(title, id);
  if (catalog.insertBook(book)) {
    return (book);
  }
  return null;
}
```

```
public boolean issue(Member member) {
  borrowedBy = member;
  dueData = new GregorianCalendar();
  dueData.setTimeInMillis(System.currentTimeMillis());
  switch (bookType) {
    case PERIODICAL:
      Calendar cutoffDate = new GregorianCalendar();
      cutoffDate.setTimeInMillis(System.currentTimeMillis());
      cutoffDate.add(Calendar.MONTH, -3);
      if (cutoffDate.after(dateAcquired)) {
        dueDate.add(Calendar.WEEK_OF_MONTH, 1);
      } else {
        return false;
      }
      break;
    default:
      dueDate.add(Calendar.MONTH, 1);
      break;
  }
  return true;
}
```

Other Book Methods

- Additionally, we would need to update `getAuthor()` and `toString`

```
public String getAuthor() {
  if (bookType == PERIODICAL) {
    return "";
  }
  return author;
}

public String toString() {
  if (bookType == BOOK) {
    return "title " + title + " author " + author + " id " + id
        + " borrowed by " + borrowedBy;
  } else {
    return "title " + title + " id " + id + " borrowed by " +
        borrowedBy + " Acquired on " + dateAcquired.getTime().toString();
  }
}
```

- We would need to make similar changes for all other `Book` methods
  - I.e., adding switching logic

For Next Time

- Review this Lecture
- Review Chapter 9.1 - 9.4
- Read Chapter 9.5 - 9.9
- Come to Class

Are there any questions?