Inheritance
The power of Object-Oriented Programming
The “canonical” example, introduced by Alan Kay
You’ll see it in Lab!
Inheritance

- Inheritance is a fundamental technique used to

- The child is a more specific version of parent

- The child **inherits** characteristics of the parent (methods and data defined by the parent class)

- Tailor derived class by **adding** new variables or methods, or by **modifying** the inherited ones

- Keyword **extends** is used to establish an inheritance (is-a) relationship

```java
class Child extends Parent {
    // class contents
}
```
Class Hierarchies

- A child class can be the parent of another child, forming a **class hierarchy**

- Two children of the same parent are called **siblings**

- Common features should be put as high in the hierarchy as is reasonable

- An inherited member is passed continually down the line
  - Therefore, a child class inherits from all its ancestor classes
Consider classes Manager and Employee. Which should be the superclass and which should be the subclass?

**Answer:** Because every manager is an employee but not the other way around, the Manager class is more specialized. It is the subclass, and Employee is the superclass.
Self-Check

What are the inheritance relationships between classes BankAccount, CheckingAccount, and SavingsAccount?

**Answer:** CheckingAccount and SavingsAccount both inherit from the more general class BankAccount.
/** Words.java
 * Demonstrates the use of an inherited method.
 * @author Java Foundations
 */

public class Words {
    /**
     * Driver: Instantiates a derived class and invokes its inherited
     * and local methods.
     */
    public static void main (String[] args) {
        Dictionary webster = new Dictionary();
        System.out.println("Number of pages: " + webster.getPages());
        System.out.println("Number of definitions: " + webster.getDefinitions());
        System.out.println("Definitions per page: " + webster.computeRatio());
    }
}

---
/**
 * Represents a book. Used as the parent of a derived class to
 * demonstrate inheritance and the use of the super reference.
 * @author Java Foundations
 */

public class Book{
    protected int pages = 1500;

    //Note: No constructor

    /**
     * Pages setter (mutator)
     * @param numPages number of pages in book
     */
    public void setPages(int numPages){
        pages = numPages;
    }

    /**
     * Pages getter (accessor)
     * @return number of pages in book
     */
    public int getPages(){
        return pages;
    }
}

Book

Dictionary

Words

-definitions : int
+computeRatio() : double
+setDefinitions(numDefinitions : int) : void
+getDefinitions() : int

main(args : String[]) : void

pages : int
+setPages(numPages : int) : void
+getPages() : int
public class Dictionary extends Book {
  private int definitions = 52500;

  /**
   * Computes ratio of definitions per page
   * @return definitions per page
   */
  public double computeRatio() {
    return definitions / pages;
  }

  /**
   * Setter (mutator)
   * @param numDefinitions number of definitions in dictionary
   */
  public void setDefinitions(int numDefinitions) {
    definitions = numDefinitions;
  }

  /**
   * Getter (accessor)
   * @return the number of definitions, an integer
   */
  public int getDefinitions() {
    return definitions;
  }
}
**The protected Modifier**

- A **protected** variable is visible to any class in the same package as the parent class.
- The **protected** modifier allows a child class to reference a variable or method directly in the parent class.
- It provides more **encapsulation** than **public** visibility, but is not as tightly encapsulated as **private** visibility.
The `super` Reference

- Constructors are **not** inherited, even though they have public visibility
  - Yet, we often want to use the parent's constructor to set up the “parent's part” of the object

- The keyword `super` can be used to refer to the parent class, including the parent's constructor

- A child’s constructor should:
  - Call the parent’s constructor as it’s first line: `super();`
  - If it does not call `super()`, a 0-parameters `super()` constructor will be called anyway!

- The `super` reference can also be used to reference other variables & methods defined in parent’s class
```java
/**
 * Demonstrates the use of an inherited method.
 * @author Java Foundations
 */

public class Words2 {
    /**
     * Driver: Instantiates a derived class and invokes its inherited
     * and local methods.
     */
    public static void main (String[] args) {
        Dictionary2 webster = new Dictionary2(1500, 52500);

        System.out.println("Number of pages: " + webster.getPages());
        System.out.println("Number of definitions: " + webster.getDefinitions());
        System.out.println("Definitions per page: " + webster.computeRatio());
    }
}
```
**

* Represents a book. Used as the parent of a derived class to
* demonstrate inheritance and the use of the super reference.
* @author Java Foundations
*/

public class Book2 {
    protected int pages;

    // Constructor (will be invoked by child)
    public Book2 (int numPages) {
        pages = numPages;
    }

    /**
     * @param number of pages in the book
     */
    public void setPages (int numPages) {
        pages = numPages;
    }

    /**
     * @return number of pages in the book
     */
    public int getPages () {
        return pages;
    }
}
/**
 * Represents a dictionary, which is a book. Used to demonstrate
 * the use of the super reference.
 * @author Java Foundations
 */

public class Dictionary2 extends Book2 {
    private int definitions;

    /**
     * Constructor. Note: Calls super
     */
    public Dictionary2(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }

    public double computeRatio() {
        return definitions / pages;
    }

    public void setDefinitions(int numDefinitions) {
        definitions = numDefinitions;
    }

    public int getDefinitions() {
        return definitions;
    }
}
Overriding

- What happens when a parent and a child class have methods with the same name?

- A child class can *override* the definition of an inherited method in favor of its own

- A method in the parent class can be invoked explicitly using the super reference, as in:
  ```
  super.message()
  ```
Thought.java

```java
/**
 * Represents a stray thought. Used as the parent of a derived
 * class to demonstrate the use of an overridden method.
 * @author Java Foundations
 */

class Thought{
    /**
     * Prints a message
     */
    public void message()
    {
        System.out.println("I feel like I'm diagonally parked in a " +
                           "parallel universe.");
    }
}
```
Advice.java

```java
/**
 * Represents some thoughtful advice. Used to demonstrate the
 * use of an overridden method.
 * @author Java Foundations
 */

public class Advice extends Thought {
    /**
     * Prints a message. Overrides the parent's version.
     */
    public void message()
    {
        System.out.println("Warning: Dates in calendar are closer" +
                          " than they appear.");
        //explicitly invokes the parent's version
        super/message();
    }
}
```
```java
/**
 * Demonstrates the use of an overridden method.
 * @author Java Foundations
 */

public class Messages {
    /**
     * Creates two objects and invokes the message method in each
     */
    public static void main(String[] args) {
        Thought parked = new Thought();
        Advice dates = new Advice();

        parked.message();
        dates.message(); // overridden
    }
}
```
Overriding

- A child class can *override* the definition of an inherited method in favor of its own

- A method in the parent class can be invoked explicitly using the super reference, as in:
  
  ```java
  super.message()
  ```

- If a method is declared with the `final` modifier, it *cannot* be overridden

- The concept of overriding can be applied to variables and is called *shadowing variables*. It is

  - Shadowing variables should be *avoided* because it tends to cause unnecessarily confusing code
Overloading vs. Overriding

- **Overloading** deals with multiple methods with the same name in the **same class**, but with **different signatures**

- **Overriding** deals with two methods, one in a **parent class** and one in a **child class**, that have the **same signature**

- Overloading lets you define a similar operation in different ways for different parameters

- Overriding lets you define a similar operation in different ways for different object types
What happens if in the class `Advice.java`, in the call `super.message();` we omit the `super`?

**Answer:** The method `message()` calls itself! An infinite loop!
Abstract classes

When you want to tell your children what to do, but you do not want to do it for them!
Need for Abstract Classes

Abstract classes and methods are shown in italics font.
Abstract Classes

- An *abstract class* is a **placeholder** in a class hierarchy that represents a **generic** concept.

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

- To declare a class as abstract:

  ```java
  public abstract class Shape {
    // contents
  }
  ```

- Abstract classes are an important element of software design: they allow us to establish **common** elements in a hierarchy that are too **generic** to instantiate.
Abstract Classes: Rules

- An abstract class often contains abstract methods with no definitions
  - The abstract modifier must be applied to each abstract method

- An abstract class typically contains non-abstract methods with full definitions

- A class declared as abstract does not have to contain abstract methods – simply declaring it as abstract makes the class abstract

- The child of an abstract class must override the abstract methods of the parent, or it, too, will be considered abstract

- An abstract method cannot be defined as final or static
The Object Class

The Mother of all Classes
The **Object Class**

- A class called Object is defined in the java.lang package of the Java standard class library
- All classes are derived from the Object class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class
- Therefore, the Object class is the ultimate root of all class hierarchies
The Object Class

- Some methods of the `Object` class:

```java
boolean equals (Object obj)
    Returns true if this object is an alias of the specified object.

String toString ()
    Returns a string representation of this object.

Object clone ()
    Creates and returns a copy of this object.
```
The **Object** Class Methods

- The *Object* class contains a few useful methods, which are inherited by all classes.

- I.e., the `toString()` method is defined in the *Object* class.

- Every time we define the `toString` method, we are actually **overriding** an inherited definition.

- The `toString` method in the *Object* class is defined to return a string that contains the name of the objects class along with some other information.

- Also in *Object*:
  - `equals()` returns `T` if and only if __________
  - `clone()` returns __________
The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases.

- We can override `equals` in any class to define equality in some more appropriate way.

- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters.

- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version.
Overriding the equals Method

equals method checks whether two objects have the same content:
```java
if (stamp1.equals(stamp2)) . .
// Contents are the same
```

== operator tests whether two references are identical - referring to the same object:
```java
if (stamp1 == stamp2) . .
// Objects are the same
```

**Figure 10** Two References to Equal Objects

**Figure 11** Two References to the Same Object
Overriding the equals Method

- To implement the equals method for a Stamp class -
  
  Override the equals method of the Object class:

  ```java
  public class Stamp {
      private String color; private int value;
      ...
      public boolean equals(Object otherObject) {
          ...
      }
  }
  ```

- Cannot change parameter type of the equals method - it must be Object

- Cast the parameter variable to the class Stamp instead:

  ```java
  Stamp other = (Stamp) otherObject;
  ```