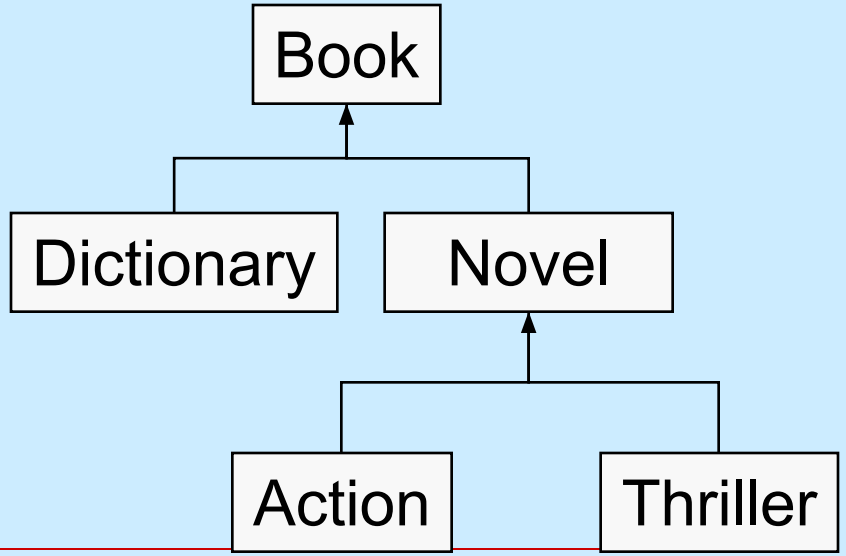
Inheritance II

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class* or *superclass*
- The derived class is called the *child class* or *subclass*.
- The subclass is a more specific version of the Original



Inheritance

- The child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Dictionary extends Book {

      // class contents
}
```

```
Dictionary webster = new Dictionary();
webster.message();
webster.defMessage();
```

Number of pages: 1500

Number of definitions: 52500

Definitions per page: 35

```
public class Book {

    protected int pages = 1500;

    public String message() {
        System.out.println("Number of pages: " + pages);
    }
}

public class Dictionary extends Book {

    private int definitions = 52500;

    public void defMessage() {
        System.out.println("Number of definitions" +
                           definitions);
        System.out.println("Definitions per page: " +
                           (definitions/pages));
    }
}
```

Some Inheritance Details

- An instance of a child class does not rely on an instance of a parent class
  - Hence we could create a Dictionary object without having to create a Book object first
- Inheritance is a one-way street
  - The Book class cannot use variables or methods declared explicitly in the Dictionary class

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

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 **`super`** reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the **`super`** reference to call the parent's constructor
- The **`super`** reference can also be used to reference other variables and methods defined in the parent's class

The keyword "super"

- It is possible to access overriding members by using the **super** keyword
- **Super** much like **this** keyword except that super doesn't refer in the current object but rather to its superclass.

Constructors of Subclasses

- Can invoke a constructor of the direct superclass.
  - super(…) must be the first statement.
  - If the super constructor call is missing, by default the no-arg super() is invoked implicitly.
- Can also invoke another constructor of the same class.
  - this(…) must be the first statement.

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }
}


public class Dictionary {

    private int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }
}
```

Example of “this” Calls

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Point() { // default constructor

    this(0,0);

  }
}
```

Example of “super” Calls

```
public class ColoredPoint extends Point {
  private Color color;

  public ColoredPoint(int x, int y, Color color) {
     super(x,y);
    this.color = color;
  }

  public ColoredPoint(int x, int y) {
    this(x, y, Color.BLACK); // point with default value
  }

  public ColoredPoint() {
    color = Color.BLACK;    // what will be the values of x and y?
  }
}
```

Default Constructor

- If no constructor is defined, the following form of no-arg default constructor is automatically generated by the compiler.

```
public ClassName() {
  super();
}
```

Execution Order of Constructors

Rule: Super class's field initializes first

Example: S x = new S();

```
public class S extends T {
  int y = 30;  // third

  public S() {
    super();
    y = 40;   // fourth
  }
  // ...
}
```

```
public class T {
  int x = 10;  // first

  public T() {
    x = 20;    // second
  }
  // ...
}
```

Overriding Methods

- When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version **overrides** the parent's version in favor of its own.
  - **Signature:** method's name along with number, type, and order of its parameters
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the **`super`** reference
- If a method is declared with the **`final`** modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding Methods (Cont.)

```
public class T {
  public void m() { … }
}

public class S extends T {
  public void m() { … }
}

T t = new T();
S s = new S();
t.m(); // invoke m of class T
s.m(); // invoke m of class S
```

Overriding Methods (Cont.)

- Dynamic dispatch (binding): The method to be invoked is determined at runtime by the runtime type of the object, not by the declared type (static type).

```
class Student {
  public int maxCredits() { return 15; }
  …
}
class GraduateStudent extends Student {
  public int maxCredits() { return 12; }
  …
}

Student s;
// …
s.getMaxCredits(); // which maxCredits method?
```

```
public class Book {

    protected int pages;

    Book(int numPages) {
        pages = numPages;
    }

    public void message() {
        System.out.println("Number of pages: " + pages);
    }
}
public class Dictionary extends Book{

    protected int definitions;

    Dictionary(int numPages, int numDefinitions) {
        super(numPages);
        definitions = numDefinitions;
    }

    public void message() {
        System.out.println("Number of definitions" +
                           definitions);
        System.out.println("Definitions per page: " +
                           (definitions/pages));
        super.message();
    }
}
```

Overloading vs. Overriding

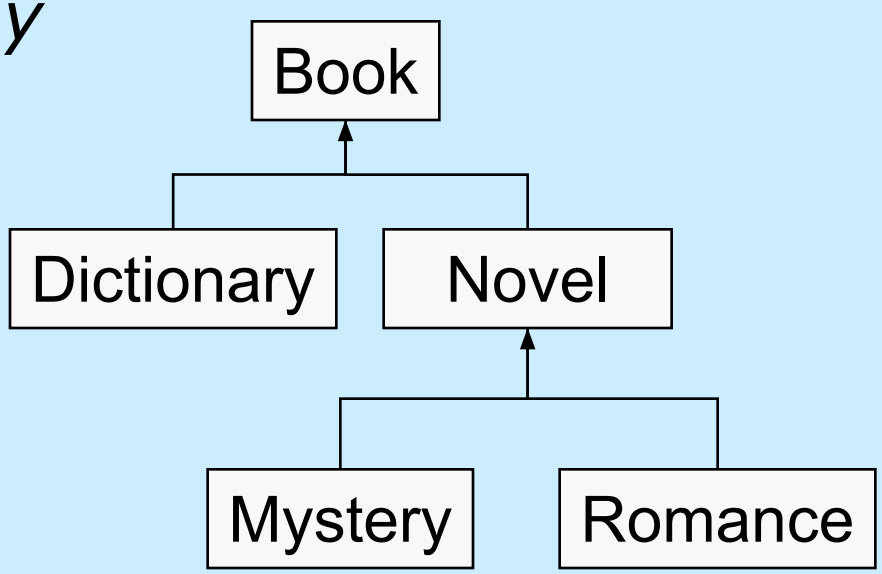
- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead (will discuss later)

Class Hierarchies

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



Class Hierarchies

- Two children of the same parent are called *siblings*
  - *However they are not related by inheritance because one is not used to derive another.*
- 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

Substitution Property

- Rules of (polymorphic) assignment
  - A Superclass Variable Can Reference a Subclass Object

```
class Student { … }
class Undergraduate extends Student { … }
class Graduate extends Student { … }

Student s1, s2;
s1 = new Undergradute(); // polymorphic assignment
s2 = new Graudate(); // polymorphic assignment

Graduate s3 = s2; // is this OK?

Graduate s3 = (Graduate) s2; // explicit casting
```

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

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information
  - All objects are guaranteed to have a `toString` method via inheritance, thus the `println` method can call `toString` for any object that is passed to it

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
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version