CS 251
Intermediate Programming
Inheritance

Brooke Chenoweth

University of New Mexico

Spring 2025

Inheritance

We don't inherit the earth from our parents,
We only borrow it from our children.

What is inheritance?

- Just as for humans, sub classes (children) inherit certain properties from their parents.
- Object Oriented - Doesn't happen if no objects
- One mother/father (`super`) object
- Child inherits all `public` and `protected` members

Inheritance

It is the same thing as having parents!

- You get part of what they have, but you can form your own way.
- If there are parents, there are usually grand-parents too. You get something from them as well...

Everything is an `Object`

- The `Object` class is the Adam/Eve of Java.
- All classes are subclasses of the `Object` class
- Therefore they inherit certain methods (such as `toString()`)
- `Object.toString()` just uses the type and hashcode of the object

Subclasses `extend` parents...

In java, all classes `extend` at least the `Object` class (implicitly).
To `extend` means to get certain properties from your parent

extends example

An example:

```
public class ParentClass {
  public void print() {
    System.out.println("Hello");
  }
}

public class ChildClass extends ParentClass {
  public void tryit() { print(); }

  public static void main(String[] argv) {
    ChildClass test = new ChildClass();
    test.tryit();
  }
}
```

ChildClass inherited print() from ParentClass!

All classes extend `Object`

If no parent class is given, parent is `Object`

```
public class Foo {
```

is equivalent to

```
public class Foo extends Object {
```

`this` and `super`

There are two keywords that are often used with inheritance. They are:

- `this` – a reference to the current object instance. Ex: to address an instance variable you may always do: `this.variable`
- `super` – a reference to the parent part of this object instance. Use super to access methods that you are overriding in the parent.
  - Note! Calls to `super()` constructors must always be the first line in your constructor (if you want to)!
  - You can also call other constructors in your object such as `this()`.

Overriding – Overloading

- A class can (as you know) define methods
- When one class inherits another, the derived class can "override" a method in the *base class*.
- Overriding methods must have the same name, same number of parameters, and same type of all parameters
- Different from overloading (where types, or number of parameters must differ)
- If an overriding method is specified as `public final void methodName()` it can not be overridden.

Overriding – Overloading cont. . .

- Methods can be both overridden and overloaded at the same time.
- Fields can not be overridden, only hidden
- Only *accessible methods* can be overridden
- Static members are always hidden (so impossible to override)

Reminder: Access modifiers

`private` Only this class

package-private No modifier. This class and others in same package.

`protected` This class and its subclasses (and same package)

`public` Accessible to all

Overriding – Annotation

- The compiler can help check that you are doing what you think you are doing.
- The `@Override` annotation tells the compiler that you intend to override a method from the superclass.
- If you don't, compiler will complain.

```
public class Bar extends Foo {
  @Override
  public int fooMethod() {
  }
}
```

Multiple types

- A subclass can be assigned to a variable of the supertype class.
- If you want to check if it's one or the other, you can use the `instanceof` keyword to check:

```
if ( obj instanceof MyClass ) {
  // if true, can safely cast here
```

- `instanceof` works with interfaces, too!

Pattern matching with `instanceof`

Let's assume with have a `Shape` interface, implemented by `Rectangle`

```
if( shape instanceof Rectangle ) {
  Rectangle r = (Rectangle)shape;
  // do Rectangle stuff here
}
```

The `instanceof` check tells us we can safely cast.

Pattern matching with `instanceof`

Let's assume with have a `Shape` interface, implemented by `Rectangle`

```
if( shape instanceof Rectangle ) {
  Rectangle r = (Rectangle)shape;
  // do Rectangle stuff here
}
```

The `instanceof` check tells us we can safely cast. As of JDK 17, we can simplify this test plus cast like so:

```
if( shape instanceof Rectangle r ) {
  // do Rectangle stuff here
}
```

Final classes and methods

Classes and methods that are defined as final cannot be extended or overridden respectively. Can be useful if you're sure that you don't want your class extended.

- Increased security
- Guarantees specific implementation
- Generally should only call `final` helpers in constructors.

Abstract Classes

- A method header without implementation is called an `abstract` method
- Any class containing an `abstract method`, must be declared as an `abstract class`
- Abstract classes can not be instantiated
- Classes extending the abstract class must provide implementation for the abstract methods.

Abstract class example

We can partially implement an abstract parent...

```
public abstract class GraphicObject {
  private int x, y;

  public void moveTo(int newX, int newY) {
    // change x, y here
  }

  public abstract void draw();
}
```

...and finish implementing in a concrete child type

```
public class Circle extends GraphicObject {
  public void draw() {
    // draw a circle
  }
}
```

Interfaces

- An interface is the extreme abstract class, containing only method headers, and no implementations at all.[1]
- The interface, is a specification of the interface between a programmer and a class
- Any class that `implements` an interface, must provide implementations for all methods in the interface (unless it's an abstract class)
- Interfaces and abstract classes are formal specifications of the interaction between classes (without implementation)

[1] Java 8 changes this a bit with *default implementations*

Example interfaces

- `Comparable` – Imposes a natural ordering of objects implementing it
- `Collection` – Common interface for all collections (implementing classes: `ArrayList`, `LinkedList`, `Vector`, etc...
- `Iterator` – Provides a serialization of collections
- `List` – Defines methods common for lists
- etc...
- Note! You are likely to have to implement some java interfaces in upcoming assignments

Abstract Classes, Interfaces, & Inheritance

- A class can only `extend` one (1) class (abstract or concrete)
- A class can implement any number of interfaces
- Abstract classes extending abstract classes do not need to implement abstract methods in the superclass. However, concrete classes extending the derived abstract class must implement all abstract methods in both superceding abstract classes.

Interfaces can inherit, too!

```
public interface Foo {
  void fooMethod();
}
```

```
public interface Bar extends Foo {
  void barMethod();
}
```

```
public class Baz implements Bar {
  public void fooMethod() {
  }
  public void barMethod() {
  }
}
```

Another inheritance example

```
public class Student {

  public String toString() {
    return "I'm a student";
  }
}
```

```
public class Undergrad extends Student {

  public String toString () {
    return super.toString() + " in college!";
  }
}
```

I'm curious...

Think of our two classes `Student` and `Undergrad`

- In the case of:

```
Student myStudent = new Undergrad();
```

- How does Java know how to execute the overridden method `toString()` in `Undergrad` when the programmer calls:

```
myStudent.toString()
```

  especially in the case where the `Student` class was compiled before `Undergrad`?

Dynamic (Late) Binding

- Compiler puts in a flag saying "Use applicable definition for method `toString()`" when compiling, since it doesn't know what definition it will use later.
- Method definition is chosen based on the current object's place in the inheritance chain, not by the type of the variable containing it!
- Even typecasts will not change this behavior:

  ```
  Student myStudent = (Student) new Undergrad();
  ```

  `myStudent.toString()` will still call the `Undergrad` `toString()` method.

Polymorphism

the quality or state of existing in or assuming different forms
In object oriented programming, the term is used to describe a variable that may refer to objects whose class is not known at compile time, and which respond at run time according to the actual class of the object to which they refer.

Definitions from `dictionary.com`

Dynamic Binding vs. Polymorphism

- Sounds like the same thing...
- That's because it is because both phrases describe the same process but from different perspectives:
  - Polymorphism is at the object level (for the programmer)
  - Dynamic Binding is the compiler's way of realizing polymorphism
- Sometimes used interchangably. Note though, original definition of polymorphism was only referring to type generality, but has been redefined for object oriented programming.

`instanceof`

- Can be used to see if instances were created from the same class.
- Useful when comparing objects - makes for a better comparison
- Syntax:
  `<object> instanceof <type>`
- Add to `equals` or `compareTo` method to ensure that there's class correspondence.

Inheritance wrapup...

- Defines relationships between categories of objects
- Allows for increasingly specialized implementations without having to reimplement (inherited methods)
- Enforced by Abstract methods and interfaces – that define the allowable use