Intuition
- Some classes have significant overlap in functionality
  - bicycles and vehicles
  - both have owners and both can move
  - students and persons
  - both have names and both can be selected for juries
  - rectangles and shapes
  - both have a color
- These are all examples of an "is a" relationship
- This is a common (but poor) intuitive litmus test
- Classes define types, which are sets of possible values

Class Hierarchy
- Inheritance is transitive
- Every class inherits from java.lang.Object
- Vocabulary
  - superclass/subclass
  - base class/derived class
  - parent class/child class

Declaration: Extends
- Keyword: extends
- public class Student extends Person {...}
  - If omitted, java.lang.Object is implicit
- Can extend one other class
  - "single inheritance" (cf C++)
- A subclass has all the members of its superclass!
  - Not the private members
  - Not the constructors (just fields and methods)
- Subclass can add new members (hence "extends")
  - New fields and new methods
  - Defines its own constructor(s)
- Subclass can modify inherited methods
  - Changes behavior
  - "overriding"

Example: Code
```java
class Person {
    private String name;
    Person () {
        name = "Baby Doe";
    }
    Person (String name) {
        this.name = name;
    }
    void rename (String name) {
        this.name = name;
    }
    String getName () {
        return name;
    }
}

class Student extends Person {
    private int ID;
    Student () {
        ID = 0;
    }
    Student (String name, int ID) {
        super(name);
        this.ID = ID;
    }
    boolean winsTicketLottery () {
        return (ID % 2 == 0);
    }
    String toString () {
        return
            getName() +
            "\[" + ID + "]";
    }
}
```

Example: Graphical View
Constructing New Instances

- Members of Student:
  - its own: ID, winsTicketLottery(), toString()
  - its parent’s: rename(), getName()
  - its parent’s parent’s: see java.lang.Object
- When a new instance is created:
  - First, the parent’s constructor is invoked
  - Can be done explicitly with super()
  - Otherwise, parent’s default constructor is called
  - Next, any initialization blocks are executed
  - Finally, the child’s constructor is executed

Static vs Dynamic Type

- Static Type = set at compile time
- Dynamic Type = set at run time

Examples

```java
Person p = new Person("Pierre");
Student s = new Undergrad("Liz", 12345);
p = s; //run-time type of p is: Person
```

Compiler can not infer dynamic type

```java
void select (Person p) {
    // static type of p is: Person
    // dynamic type of p is: ???
    . . .
}
```

Operator instanceof tests the run-time type

```java
if (p instanceof Faculty) { ... }
else if (p instanceof Student) { ... }
```

Narrowing vs Widening

- Recall basic types (eg long, int)
- Widening
  - Assign a "small" value to a variable of "big" type
  - This is always ok and so can be done implicitly
    ```java
    int i = 45;
    long x = i; //ok
    ```
- Narrowing
  - Assign a "big" value to a variable of "small" type
  - The correctness of this cannot be checked by compiler and so requires an explicit cast
    ```java
    long x = 45;
    int i = (int)x; //ok? compiler promise
    ```

Narrowing and Widening Objects

- Subclasses are “smaller” types than superclasses

```
Vehicle
Bicycle
```

```
Every bicycle is a vehicle
```

Narrowing and Widening Objects

- Widening
  - Assign a subclass (static type) to a variable of superclass (static) type
  - This is always ok and so can be done implicitly
    ```java
    Student s = new Student();
    Person p = s; //ok
    ```
- Narrowing
  - Assign a superclass (static type) to a variable of subclass (static) type
  - This cannot be checked by the compiler and so requires an explicit cast
    ```java
    Person p = new Student();
    Student s = p; //compiler complains
    Student s = (Student)p; //ok? program promise
    ```

Argument Passing

- Method argument types must match signature
- Automatic (implicit) widening
  ```java
  Student s = ...;
  cse459.enroll(s); //ok (automatic widening)
  ```
- Cast for (explicit) narrowing
  ```java
  Person p = ...;
  someJury.select(p); //ok
  ```

```java
CSE459.enroll(p); //compiler complains
```
Overriding Methods

- Overriding: a subclass declares a method that is already present in its superclass.
- Note: signatures must match (otherwise it is just overloading).

```java
class Person {
    String toString() {
        return getName();
    }
}

class Student extends Person {
    String toString() {
        return getName() + " [" + ID + "]";
    }
}
```

- Question: which method is called?
  ```java
  Person p = new Student();
  System.out.println(p.toString());
  ```

Overriding: Graphical View

Best Practices: @Override

- Use `@Override` annotation with all methods intended to override a method in a superclass.

```java
class Student extends Person {
    @Override
    String toString() {
        // ... 
    }
}
```

- Compiler complains if there is no matching method in superclass.
- Prevents accidental overloading if a mistake is made in the signature.
- We (may) talk about annotations later...

Polymorphism

- Answer: The dynamic type determines which method is called.
  ```java
  Person p = new Student();
  p.toString(); // calls Student version
  ```

- Informal model:
  - Method invocation is a run-time message to the object.
  - That (run-time) object receives the request, performs the action, and returns the result.

- Goal: we get the right behavior regardless of actual (run-time) type.

```java
Person[] csePeople = ... // students & faculty in CSE
for (int i = 0; i < csePeople.length; i++) {
    csePeople[i].toString();
}
```

- Note: this applies to methods only, not fields.
  - Fields can not be overridden, only hidden.

Hook methods

- Dynamic type of this controls which method executes.
- Hook method: Called internally, intended to be overridden.

```java
class Course {
    void enroll(Student s) {
        if (checkEligibility(s)) { ... }
    }
    boolean checkEligibility(Student s) {
        // determines whether s has prereqs for this course
    }
}
class Tutorial extends Course {
    boolean checkEligibility(Student s) {
        // determines whether s has paid
    }
}
```

- Yo-yo problem:
  - Must trace up & down class hierarchy to understand code.

```java
Course workshop = new Tutorial();
workshop.enroll(s);
```

Protected

- We’ve seen three levels of visibility:
  - `private`: concrete representation.
  - `default`: trusted and collocated.
  - `public`: abstract interface to all clients.
- Writing a subclass often requires:
  - More access than a generic client.
  - Less access than whole concrete representation.
- Solution: new visibility level.
  - Keyword: `protected`.
  - Protected members are inherited but are not part of the public interface to generic clients.
  - Warning: anyone can extend your class and get access to protected members.