CS1410

Midterm 1 Prep

Strings

Class `String`

- Strings are **immutable**

  i.e. after a String has been created the character contents can no longer be changed

Comparing Strings

- **==**

  compares instance (address of the object)

- **equals:**

  compares content of String objects

- **equalsIgnoreCase:**

  compares content disregarding upper / lower case

Comparing Strings

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

String implements Comparable<String>

Comparing Strings

- **Interface Comparable<T>**

Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

➡ String has a method compareTo

< 0 … calling string smaller than parameter

0 … calling string equal to parameter

> 0 … calling string greater than parameter

StringBuilder

Class **StringBuilder**

- Used for creating and manipulating *dynamic* string information (i.e. when strings need to be changed)

- **capacity** … specifies how many characters can be stored
- If a StringBuilder‘s capacity is exceeded, the capacity expands to accommodate additional characters.

String vs StringBuilder

- Java can perform certain **optimizations** involving String objects because Stirngs are immutable (don't change)

  ➡ **String** should be used if the data does not change

- If a program uses frequent **concatenations** or other String **modifications** **StringBuilder** is often more efficient

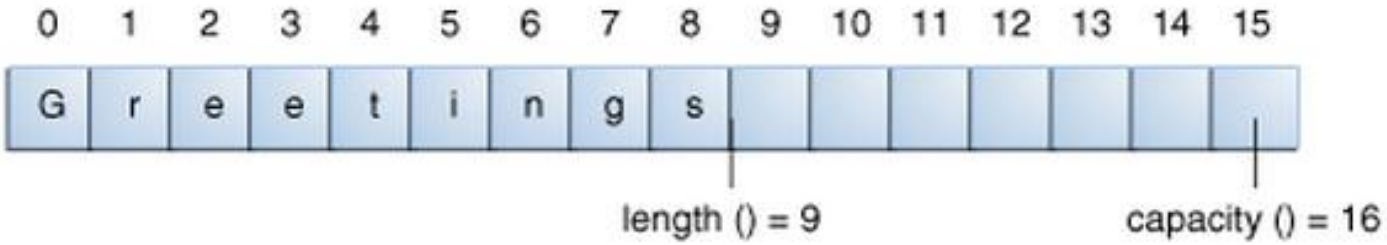
FYI: StringBuilder vs StringBuffer

Both provide the same functionality but

- **StringBuilder** is NOT thread safe
- **StringBuffer** is thread safe

Example:

StringBuilder sb = new StringBuilder("Greetings");



16.4.4 StringBuilder append Methods

StringBuilder	append(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuilder	append(char c) Appends the string representation of the char argument to this sequence.
StringBuilder	append(char[] str) Appends the string representation of the char array argument to this sequence.
StringBuilder	append(char[] str, int offset, int len) Appends the string representation of a subarray of the char array argument to this sequence.
StringBuilder	append(CharSequence s) Appends the specified character sequence to this Appendable.
StringBuilder	append(CharSequence s, int start, int end) Appends a subsequence of the specified CharSequence to this sequence.
StringBuilder	append(double d) Appends the string representation of the double argument to this sequence.
StringBuilder	append(float f) Appends the string representation of the float argument to this sequence.
StringBuilder	append(int i) Appends the string representation of the int argument to this sequence.
StringBuilder	append(long lng) Appends the string representation of the long argument to this sequence.
StringBuilder	append(Object obj) Appends the string representation of the Object argument.
StringBuilder	append(String str) Appends the specified string to this character sequence.
StringBuilder	append(StringBuffer sb) Appends the specified StringBuffer to this sequence.

Overloaded versions for primitive types, String, char arrays, Object, . . .

The compiler can use append to implement the + and += String concatenation operators

Can be chained e.g.: `myStringBuilder.append("this").append("that");`

Exercise
String

- Inheritance (Ch 9)

Introduction

Inheritance is a form of **software reuse** in which a new class is created by **absorbing an existing class's members** and embellishing them with **new or modified capabilities**.

Has-a vs Is-a relationship

- *Has-a relationship* represents composition
  - an object contains as members (fields) references to other objects
  - Examples:

    a student has-an address

    a polygon has-a number of points

    a college has-a number of teachers and it has-an address

Has-a vs Is-a relationship

- *Is-a relationship* represents inheritance
  - an object of a subclass can be treated as an object of its superclass
  - Examples:

    a student is-a person

    a polygon is-a twoDimensionalShape

    a college is-an educationalInstitution

Subclass Superclass

TODO:

Which one is NOT a superclass/subclass relationship?

a) toy – doll, board game, tinker toy
b) insect – fly, wasp, moth, bee
c) violin – viola, cello, double bass
d) mountain – K2, Nanga Parbat, Mt. Everest

TODO:

Which one is NOT a superclass/subclass relationship?

a) toy – doll, board game, tinker toy
b) insect – fly, wasp, moth, bee
c) violin – viola, cello, double bass ✓
d) mountain – K2, Nanga Parbat, Mt. Everest

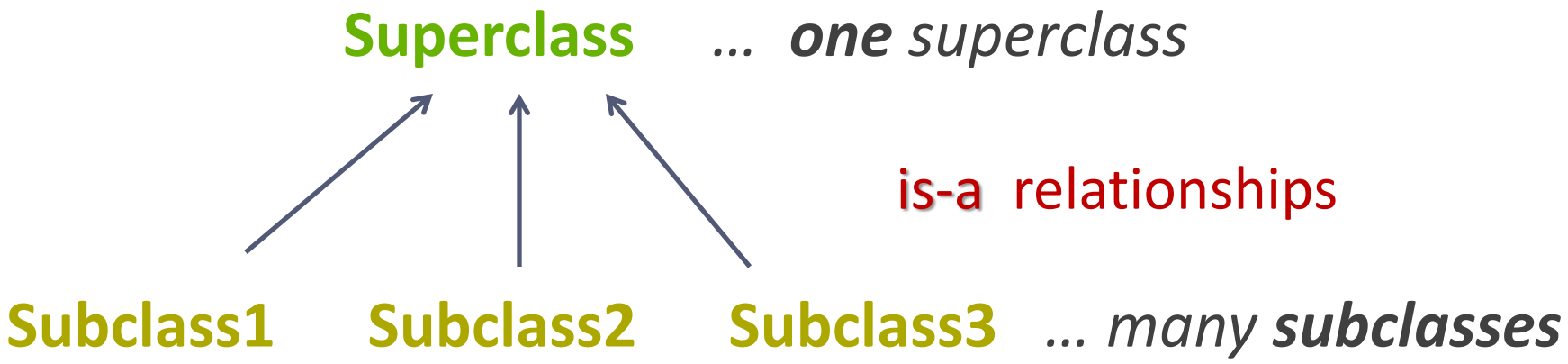
Superclasses and Subclasses

Superclass *… tends to be more general*

is-a relationship

Subclass *… tends to be more specific*

Superclasses and Subclasses



One superclass can have multiple subclasses

Each subclass has **exactly one direct** superclass

Single Inheritance

Subclass a.k.a derived class, extended class

- Subclasses inherit all the public and protected members (fields, methods, and nested classes) from its superclass.

  Constructors are not members, so they are not inherited .

Superclass a.k.a base class

- Superclass is the class from which the subclasses are derived
- If no superclass is specified the class implicitly derives from Object.
- The class Object sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class inherits the instance methods of Object

Except object

`protected` Members

Java Access Modifiers:

- `public` … accessible to any class
- `private` … accessible only within the class itself.
- `protected` … accessible within its own class, within its subclasses and within other classes in the same package

Methods, fields, .. retain their original access modifier when they are inherited by the subclass.

`protected` Members

- Useful to restrict access of methods
- BUT : fields should remain private (as a rule of thumb)

  Why?

  - 1)
  - 2)
  - 3)

`protected` Members

- Useful to restrict access of methods
- BUT： fields should remain private (as a rule of thumb)

  Why?
  - for flexibility (allows to change implementation details)
  - for control (input validation, read-only)
  - for data integrity (any subclass can modify protected fields)

Subclasses Constructors

- Constructors are **not** inherited.
- The first task of subclass constructor: call its direct superclass's constructor explicitly or implicitly

Subclasses Constructors

If the superclass does not have a default or no-argument constructor, the superclass constructor needs to be called explicitly.

Super-class constructor call syntax:

`super` + argument list of super-class constructor

```
super ( . . );
```

- must be the first statement in subclass constructor

Instantiating a subclass object begins a chain of constructor calls

@Override

Overridden methods have the exactly same signature as the method they override

Always use the annotation @Override when you modify an inherited method in a derived class

TODO: Advantages:

@Override

Overridden methods have the exactly same signature as the method they override

Always use the annotation @Override when you modify an inherited method in a derived class

Advantages:

- Catches errors

  e.g. misspelled name, incorrect parameter list

- Makes your code more clear to programmers

Keyword super

```
@Override
public String toString()
{
    return super.toString();
}
```

Superclass method can be accessed with

super + dot operator + method call

instanceof Operator

- The instanceOf operator tests whether a particular object is an instance of a given type.
- For example:

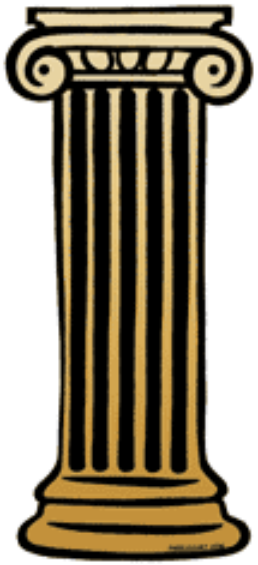
instanceof is a long name for an operator otherwise the syntax is similar to **a + b**

```
if (obj instanceof Dog)
{
     Dog myDog = (Dog)obj;
 }
```

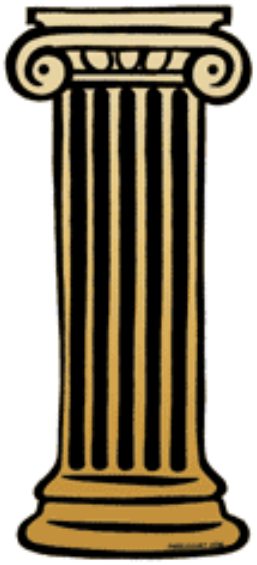
- Polymorphism (ch10)

TODO: 3 Pillars of Object Oriented Programming

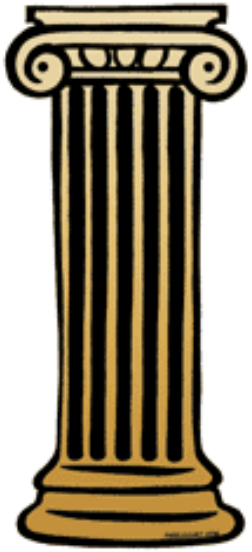
TODO: 3 Pillars of Object Oriented Programming



Encapsulation



Inheritance



Polymorphism

Polymorphism

Polymorphism is the characteristic of being able to assign a **different meaning** or usage to something **in different contexts** - specifically, to allow an entity such as a variable, a method, or an object to have more than one form.

Polymorphism Examples

Polymorphism promotes extensibility:

Software that invokes polymorphic behavior is independent of the object types to which messages are sent.

New object types that can respond to existing method calls can be added into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Polymorphism Examples

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics.

You can command objects to behave in manners appropriate to those objects, without even knowing their types - as long as the objects belong to the same inheritance hierarchy.

Polymorphism allows for specifics to be dealt with during runtime (execution)

Polymorphic Behavior

Virtual Method Invocation

At run time

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable, **NOT** the method defined by the variable's type.

The type of the referenced object, **NOT** the type of the variable, determines which method is called.

Demo Polymorphic Behavior

Subclass reference can be assigned to a superclass variable but not the other way around.

```
Animal animal = new Fish();
```

That works.
Every fish is an animal

```
Fish fish = new Animal();
```

That does NOT work.
Not every animal is a fish

ALSO: Animal does not have the fish specific class members

Downcasting

Suppose ClassB derives from ClassA

And varA is a declared and assigned instance of ClassA

Downcasting:

Treat instance of ClassA as instance of ClassB

```
ClassB varB = (ClassB) varA;
```

must be done explicitly (it works only sometimes)

Downcasting Examples

1.
```
ClassA varA = new ClassB (); // ClassB extends ClassA
ClassB varB = (ClassB) varA;
```
Works – varA is an instance of ClassB

2.
```
ClassA varA = new ClassC();  // ClassC extends ClassA
ClassB varB = (ClassB) varA;
```
Fails – varA is NO instance of ClassB

3.
```
ClassA varA = getSomeClassAObject();
ClassB varB = (ClassB) varA;
```
Might or might not work

Assigning a subclass reference to a superclass variable:

```
e.g. ClassA varA = varB;
```

- Superclass variable can access only superclass members. Attempt to access subclass-only members => compile error
- Overridden methods invoke the subclass implementation (polymorphism)
- All calls to overridden methods are resolved at run-time. (dynamic binding)

Assigning a superclass reference to a subclass variable

the superclass reference must be explicitly downcast to the subclass type

```
ClassB varB = (ClassB) varA;
```

- If varA is not an instance of ClassB an exception will occur at runtime.
- The instanceof operator can be used to ensure that varA is an instance of ClassB

Downcasting

You can use the **instanceof** operator to ensure that the instance is of the right type

```
ClassA varA = getSomeClassAObject();
if (varA instanceof ClassB)
      ClassB varB = (ClassB) varA;
```

Abstract Classes

- Used as superclass

  specify what is common among subclasses
- Cannot be instantiated (methods have no implementation)
- Can but need not include abstract methods
- Can include methods that are **not** abstract.
- Can include both static and instance fields (data)
- You can declare variables of an abstract class type

Concrete Classes

- Can be used to instantiate objects
- Provide implementations for every method they declare and for every abstract method they inherit (implement all “missing pieces”)
- If a subclass does NOT implement all abstract members of the superclass, it has to be declared abstract, too.

Abstract Methods

Intended to be overridden by subclasses

- Abstract method:
  - Declared with keyword `abstract`
  - Has no method body (no implementation)
  - Must be declared in an abstact class

  e.g.: `public abstract void draw();`

`;` instead of method implementation

- Abstract class:
  - declared with keyword `abstract`.
- Ctors and static methods canNOT be declared abstract.

getClass method

- Every object in Java knows its own class and can access this information through the getClass method
  - getClass returns an object of type Class (java.lang)
  - You can get the object's class name by invoking getName (package name + file name) or getSimpleName (only file name) on the Class object

Interfaces

10.7 Interfaces

- Interfaces form a **contract** between the class and the outside world, and this contract is enforced at build time by the compiler.
- If your class claims to implement an interface, **all methods defined** by that interface must appear in its source code before the class will successfully compile.

- Interfaces are not part of the class hierarchy
- Java has single inheritance but implementing multiple interfaces provides an alternative.
- Objects can have **multiple types**
  - The type of their own class
  - The types of all the interfaces they implement

Why should I use an interface

- They allow objects of unrelated classes to be processed **polymorphically**
- Using the same signature for the same functionality makes your code more clear and easier to read
- Share constants.

Java Interfaces

Java interfaces are reference types. They contain **NO implementation**, only

1. Constants
2. Method Signatures
3. Nested Types … e.g. enums

fields are **implicitly public, static** and **final**

methods are **implicitly public** and **abstract**

All interface members are public

Java Interfaces

- **Cannot be instantiated**
- Can be implemented by classes
- Can be extended by other interfaces

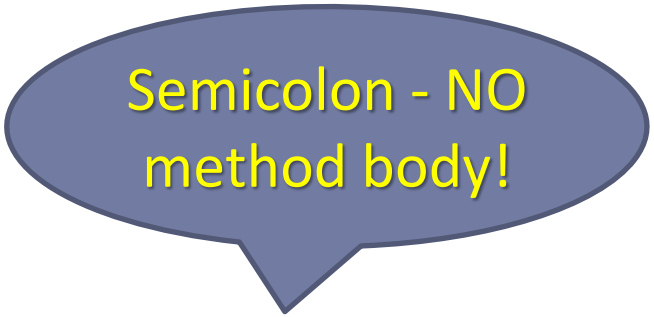
Using Interfaces

1. **Declare** an interface
2. **Implement** this interface
3. User interface **as a type**

1 How to **declare** an interface?

e.g.

```
public interface MyInterface
{
    // method signatures
    int calculateThis (int x1, int x2);
    double calculateThat (double a);
}
```



2 How to **implement** an interface?

```
e.g.:
public class ClassName implements MyInterface
```

Interface name

Write implementation of each method declared in the interface - or declare your class abstract.

3 User interface as a type

- Can use interface names anywhere you can use any other data type name.

- E.g.:

Implements Relatable

```
Relatable myRelatable = new RectanglePlus();
```

Interface type

Interface and **UML**

- Interface place `«interface»` above the interface name.
- **dashed arrow** with a **hollow arrowhead** pointing from the implementing class to the interface.
- Subclasses inherits its superclass's is-a relationships with an interface.

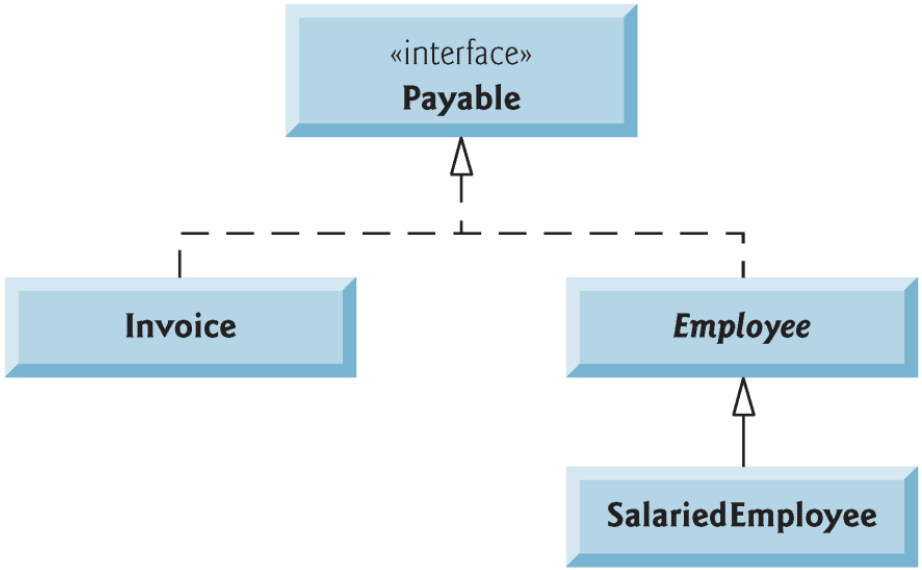


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

TODO:

TRUE

Can an object of type SalariedEmployee be added to an array of type Payable?

```
Payable[]  array = { Invoice, Employee, SalariedEmployee };
```

T H E E N D