Data Structures and Object-Oriented Design

Spring 2014
Carola Wenk
Other Data Types/Structures

• We’ve seen that the actual implementation of the data type only matters in the overall performance (and possibly functionality).

```java
class BinarySearchTree {
    public BinarySearchTree() {...}
}
```

What operations did binary search trees offer us, and how did it differ in implementation or functionality?
Other Data Types/Structures

- We’ve seen that the actual implementation of the data type only matters in the overall performance (and possibly functionality).

```java
class BinarySearchTree {
    public BinarySearchTree() {...}
    public void add(int x) {...}
    public void remove(int x) {...}
    public boolean find(int x) {...}
}
```

Do we even need to name this class to refer to its data structure?
Other Data Types/Structures

• We’ve seen that the actual implementation of the data type only matters in the overall performance (and possibly functionality).

```java
class OrderedCollection {
    public OrderedCollection() {...}
    public void add(int x) {...}
    public void remove(int x) {...}
    public boolean find(int x) {...}
}
```

Do we even need to name this class to refer to its data structure? Not really - the user doesn’t need to know how the data is organized.
What about Type Compatibility?

- So far, our class definitions have been defined to manipulate a single type (usually \texttt{int}).
- Do we really have to define a different class for a stack of strings? Can we define a general-purpose stack?

```java
class intStack {
    private int[] S = null;
    private int top;

    public intStack(int capacity) {
        S = new int[capacity];
        top = capacity;
    }

    public int pop() {
        return S[top++];
    }

    public void push(int x) {
        S[--top] = x;
    }
}

class StringStack {
    private String[] S = null;
    private int top;

    public StringStack(int capacity) {
        S = new String[capacity];
        top = capacity;
    }

    public String pop() {
        return S[top++];
    }

    public void push(String x) {
        S[--top] = x;
    }
}
```
Object-Oriented Design

- In Java, “everything is an object” and different classes can be defined to be compatible according to functionality.

```java
class A {
    ...
}

class B extends A {
    ...
}
```
Object-Oriented Design

The best way to think of type compatibility is that it is always acceptable to extend functionality, but never ok to remove it.

```java
class B extends A {
    public void g() {...}
}

class A {
    public void f() {...}
    ...
}
```

```java
A x = new A();
B y = new B();
x.f();
y.f();
y.g();
x = new B();
    // not allowed!
x.g();
x.f();
```
Object-Oriented Design

Java’s type checking is simple: a reference must “hold” at least as much functionality as it was declared to (more is ok).

```java
public class B extends A {
  public void g() {...}
}

class A {
  public void f() {...}
}

A x = new A();
B y = new B();
x.f();
y.f();
y.g();
x = new B();
// not allowed!
x.g();
x.f();
```
• The class extending functionality is called a **subclass**, and the class being extended is called the **superclass**. We can access inherited attributes using the **super** keyword.
• The class extending functionality is called a subclass, and the class being extended is called the superclass. We can access inherited attributes using the super keyword.
Rules of Inheritance

- The class extending functionality is called a **subclass**, and the class being extended is called the **superclass**. We can access inherited attributes using the `super` keyword.
Object-Oriented Design

The fancy name for how references in Java work is **type polymorphism**.

```java
class A {
    public void f() {...}
}

class B extends A {
    public void g() {...}
}

A x = new A(1);
B y = new B(1, 2.0);

x.f(1);
y.f(1);
y.g();
x = new B();
x.g();
x.f();
```
These restrictions on references allow us to check for type violations at compile-time - why is this important?

class B extends A {
    public void g() {...}
}

class A {
    public void f() {...}
}

A x = new A(1);
B y = new B(1, 2.0);
x.f(1);
y.f(1);
y.g();
x = new B();
x.g();
x.f();
**protected access**

- Any attributes that are declared protected are accessible by subclasses, but not the “outside world.”

```java
class A {
    protected void f() { ... }
    ...  
}
class B extends A {
    public void g() { ... }
    ...  
}
```

```java
A x = new A();
B y = new B();
x.f();
// not allowed!
y.f();
y.g();
x = new B();
x.g();
```
Java Access Rules

World

Subclass

Package

class

private  <none>  protected  public
We can be flexible about how we assign objects, as long as these assignments respect the defined hierarchy of compatibility:
Big Picture

We can be flexible about how we assign objects, as long as these assignments respect the defined hierarchy of compatibility:
We can be flexible about how we assign objects, as long as these assignments respect the defined hierarchy of compatibility:
Using Inheritance

- Note that references are essentially “unidirectional.”

- How general-purpose can we make types using Java’s object model?

```java
class Stack {
    private Object[] S = null;
    private int top;

    public Stack(int capacity) {
        S = new Object[capacity];
        top = capacity;
    }

    public Object pop() {
        return S[top++];
    }

    public void push(Object x) {
        S[--top] = x;
    }
}
```
Limitations

• Inheritance is useful for extending functionality, but it can’t do everything.

• By defining `Stack` to hold `Objects`, we “lose” functionality when we remove things from the stack:

```java
Stack S = new Stack(10);
S.push(new Integer(15));
S.push(new String("foo"));

// this is the only legal way to
// retrieve items - why?
Object a = S.pop();
Object b = S.pop();

// what are the types of a and b?
```
Type Casting

• Java actually allows us to regain functionality by “casting” the returned `Object` into the “correct” type.

• This helps us use one class declaration to create different kinds of `Stacks`, but does not allow a heterogeneous `Stack`.

```java
... 
Stack S = new Stack(10);
S.push(new Integer(15));
S.push(new String(“foo”));

// this is the only legal way to
// retrieve items - why?
Integer a = (Integer) S.pop();
String b = (String) S.pop();

// what are the types of a and b?
```
Java Generics

- Java also provides a mechanism to make classes generic, which avoids the need for casting:

```java
class MyClass<T> {
    private T member_variable;

    public T foo(T x) {
        ...
    }
}
```

- This way, we can use the same class definition for multiple types (without losing functionality), and errors in type usage can still be caught at compile-time.
Java Generics

- Java also provides a mechanism to make classes generic, which avoids the need for casting:

```java
class MyClass<T> {
    private T member_variable;

    public T foo(T x) {
        
    }
}
```

- Given the way Java expects us to declare everything up front - is there a potential problem with using generic types?
Another Problem

Specialized classes can implement similar functionality - but our rules (so far) for references don’t allow us to refer to such instances interchangeably.

What if these classes implement some similar functionality?
Java Interfaces

- We can specify that a Java class implements a particular kind of functionality defined as an interface.

```java
interface Collection {
    boolean add(Object o);
    boolean remove(Object o);
    boolean contains(Object o);
    boolean equals(Object o);
}
```

class Foo implements Collection {
    ...
}

class Bar implements Collection {
    ...
}

Foo X = new Foo();
Bar Y = new Bar();

Collection C;
C = X;
C = Y;

Interfaces in Java can be extended like classes, and follow the same inheritance rules.
Recap: Object-Oriented Design

- In Java, everything is an “Object” - what does this mean?
- What are the rules of inheritance for class attributes?
- What are the rules for declaring and using references to class instances?
- What are the differences between generic types and polymorphic types?
- What gap in the object-oriented paradigm do interfaces help address?