Lecture 1
Abstract Data Types and Algorithms
in Object-Oriented Style

Outline:

- OO and Java Concepts Review
- Stacks revisited OO style
- New concepts: Nested classes, Generics

Objects and Classes

- Every **object** is an **instance** of a class, which defines its **type**.
- **Types in Java** are either **base types** (e.g., int, float, double) or **class types** (**reference types**).
- A class is a **blueprint** defining an object's data (instance variables) and methods for accessing/modifying it.
- The **members** of a class are:
  - **Instance Variables** (**fields**): store an object's data.
  - **Methods**: are blocks of code that perform actions (similar to functions in other languages).

Creating and Using Objects

- In Java, base-type variables differ from class-type (reference) variables.
- Declaring a base-type variable like `int n;` allocates memory for it.
- **Declaring a class-type variable like `Apple a;` only establishes `a` as a reference variable, but does not create the object nor does it allocate memory for the object.**
- A reference variable stores the memory address of an object, and can be assigned either to an existing instance or a newly created one.
- Reference variables can also hold the value null, indicating no object.
- **Objects (instances) can be created only by using the new operator.**

Object Instances and References

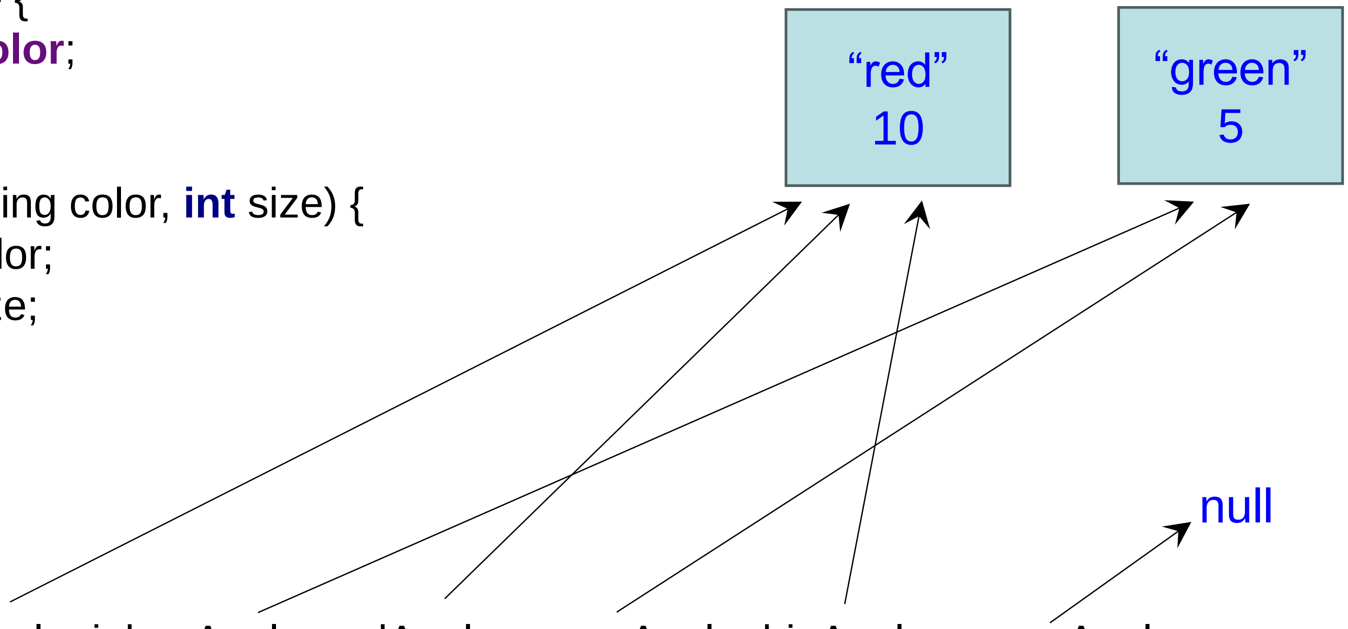
```
public class Apple {
   private String color;
   private int size;

   public Apple(String color, int size) {
      this.color=color;
      this.size = size;
   }
}
```

How many apples are there ?

```
Apple myApple, johnsApple, redApple, greenApple, bigApple, marysApple;
myApple = new Apple("red", 10);
johnsApple = new Apple("green", 5);
redApple = myApple;
greenApple = johnsApple;
bigApple = myApple;
```

Object Instances and References

```
public class Apple {
   private String color;
   private int size;

   public Apple(String color, int size) {
      this.color=color;
      this.size = size;
   }
}
```



```
Apple myApple, johnsApple, redApple, greenApple, bigApple, marysApple;
myApple = new Apple("red", 10);
johnsApple = new Apple("green", 5);
redApple = myApple;
greenApple = johnsApple;
bigApple = myApple;
```

Warmup Example Problem: Matching Parentheses

- Arithmetic expressions may contain various pairs of grouping symbols, such as:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”
- Each opening symbol must match its corresponding closing symbol!
- Examples: (between brackets can be any other symbols, omitted here)
  - Correct: ( ) (( )) ([( )])
  - Correct: ((( ) (( )) { ([( )]) }))
  - Incorrect:) (( )) { ([( )]) }
  - Incorrect: ({ [ ]) }
  - Incorrect: {
- Find out if a given expression contains correctly matched parentheses

Solution Idea

- Arithmetic expression is a string of tokens (characters)
- ***Use a stack of characters***
- Scan the string of tokens in a single left-to-right scan:
  - Each time we encounter an opening symbol, we push that symbol onto the stack
  - Each time we encounter a closing symbol, we pop a symbol from the stack (assuming it is not empty) and check that these two symbols form a valid pair.
- Expression is correct(properly matched): If we reach the end of the expression and the stack is empty [( ) (( ))] { }
- Expression is not correct:
  - We reach the end of the expression and the stack is not empty => missing right delimiter ( ) [{ }
  - During the scan, we encounter a closing symbol and the stack is empty => missing left delimiter ( ) } [ ]
  - During the scan, we encounter a closing symbol and the stack is not empty but the current symbol and the popped symbol do not match => mismatched delimiters ( ) [}]

Solution Implementation

- Stack of Characters
  - Abstract Data Type (ADT) Stack
    - Operations: push, pop, isEmpty
  - Could have various implementations:
    - Fixed size array
    - Resizeable array
    - Linked list
- Parentheses Matcher
  - Uses a stack
  - It is a Client of the Stack ADT

Parentheses Matching

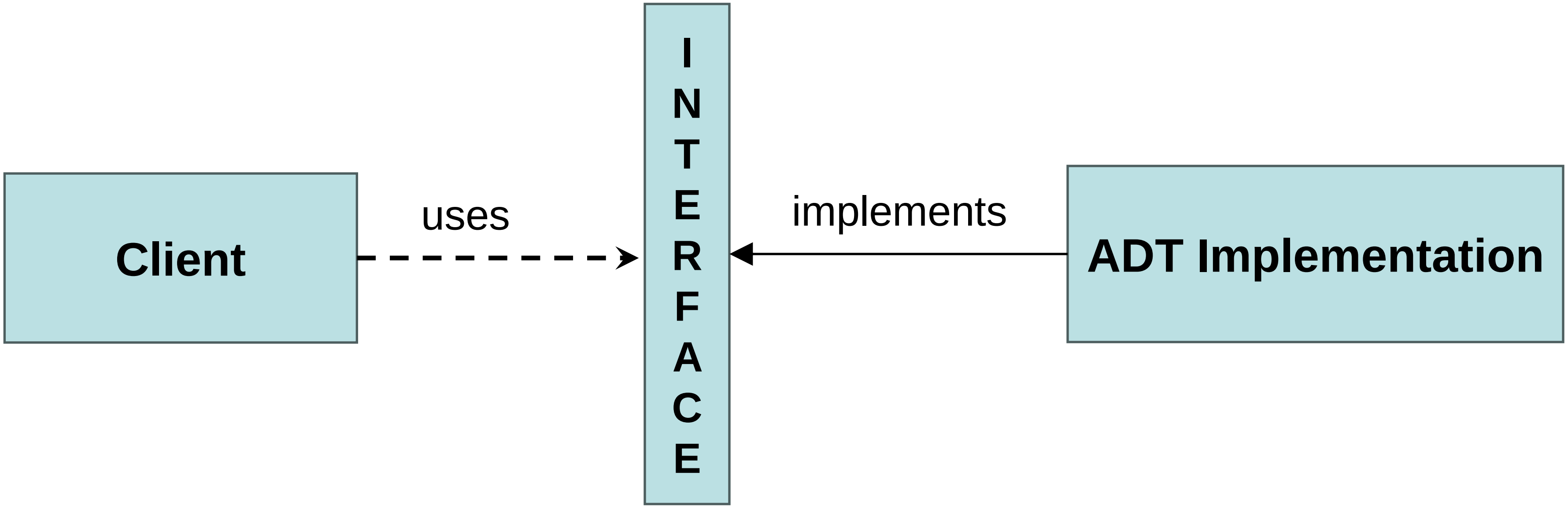
```
public class MatchingParentheses {

    public static boolean isMatched(String expression) {
// ….
}

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Input some lines containing parentheses ...");
        while (sc.hasNext()) {
            String line = sc.nextLine();
            if (isMatched(line)) System.out.println("Line is OK");
            else System.out.println("Line is NOT OK");
        }
        sc.close();
    }
}
```

Stack Client – Parentheses Matching

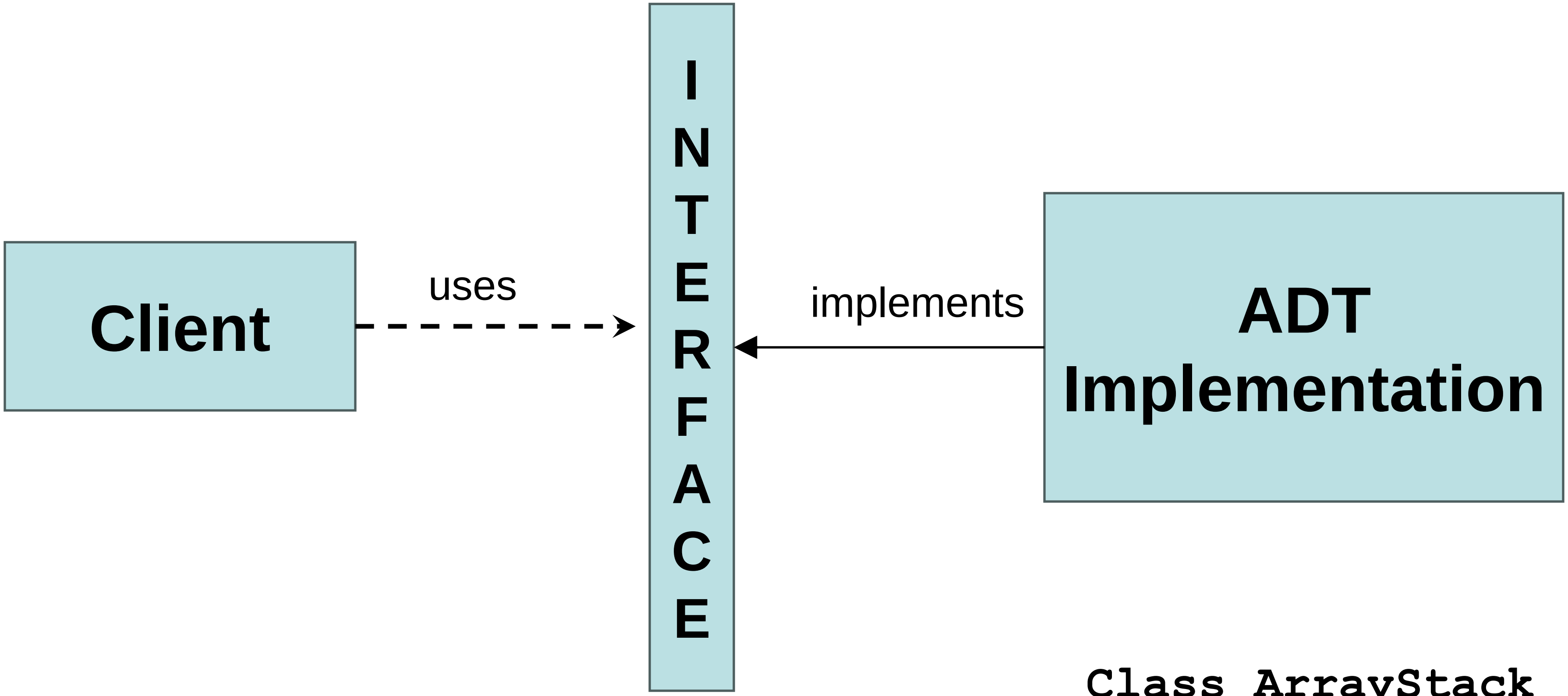
```
public static boolean isMatched(String expression) {
    final String opening = "({["; // opening delimiters
    final String closing = ")}]"; // respective closing delimiters
    StackOfChars stack = … // we need a stack implementation!
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1) // this is a left delimiter
            stack.push(c);
        else if (closing.indexOf(c) != -1) { // this is a right delimiter
            if (stack.isEmpty()) // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(stack.pop()))  // mismatched
                return false;
        }
    }
    return stack.isEmpty(); // were all opening delimiters matched?
}
```

Client-Interface-Implementation

- Application programming interface (API): description of abstract data type, basic operations with an informal description of the effect of each operation
  - In Java (and other OO languages) the *main structural element that enforces an API is an **interface**.*
- Implementation: actual code implementing operations.
- Client: program using operations defined in interface.



Client-Interface-Implementation



Client

uses

INTERFACE

implements

ADT Implementation

`ParenthesesMatcher`

`Interface StackOfChars`

`Class ArrayStack`

`Class ListStack`

Abstract Data Types and Algorithms
in Object-Oriented Style

Outline:

- OO and Java Concepts Review
- Stacks revisited OO style
- New concepts: Nested classes, Generics

The Stack API

- Simple case – A Stack of characters

```
public interface StackOfChars {

   void push(char item);    // insert a new character  onto stack

   char pop();   // removes and returns character on top of stack

   boolean isEmpty();  // test if stack is empty

}
```

Stack Implementations

- Various implementation methods:
  - Stack implemented by fixed-size array

    ```
    class ArrayStackOfChars implements StackOfChars
    ```

  - Stack implemented by resizeable array

    ```
    class ResizeableArrayStackOfChars implements StackOfChars
    ```

  - Stack implemented by linked list

    ```
    class ListStackOfChars implements StackOfChars
    ```

```
public class ArrayStackOfChars implements StackOfChars {
    private final char[] a;  // holds the items
    private int n;       // number of items in stack

    public ArrayStackOfChars(int capacity) {
        a = new char[capacity];
        n = 0;
    }

    public void push(char item) {
        if (isFull()) throw new RuntimeException("Stack overflow");
        a[n++] = item;
    }

    public char pop() {
        if (isEmpty()) throw new RuntimeException("Stack underflow");
        charr item = a[--n];
        return item;
    }

   public boolean isEmpty() {  return n == 0;  }

    private boolean isFull() {   return n == a.length;  }
}
```

Stack Discussion

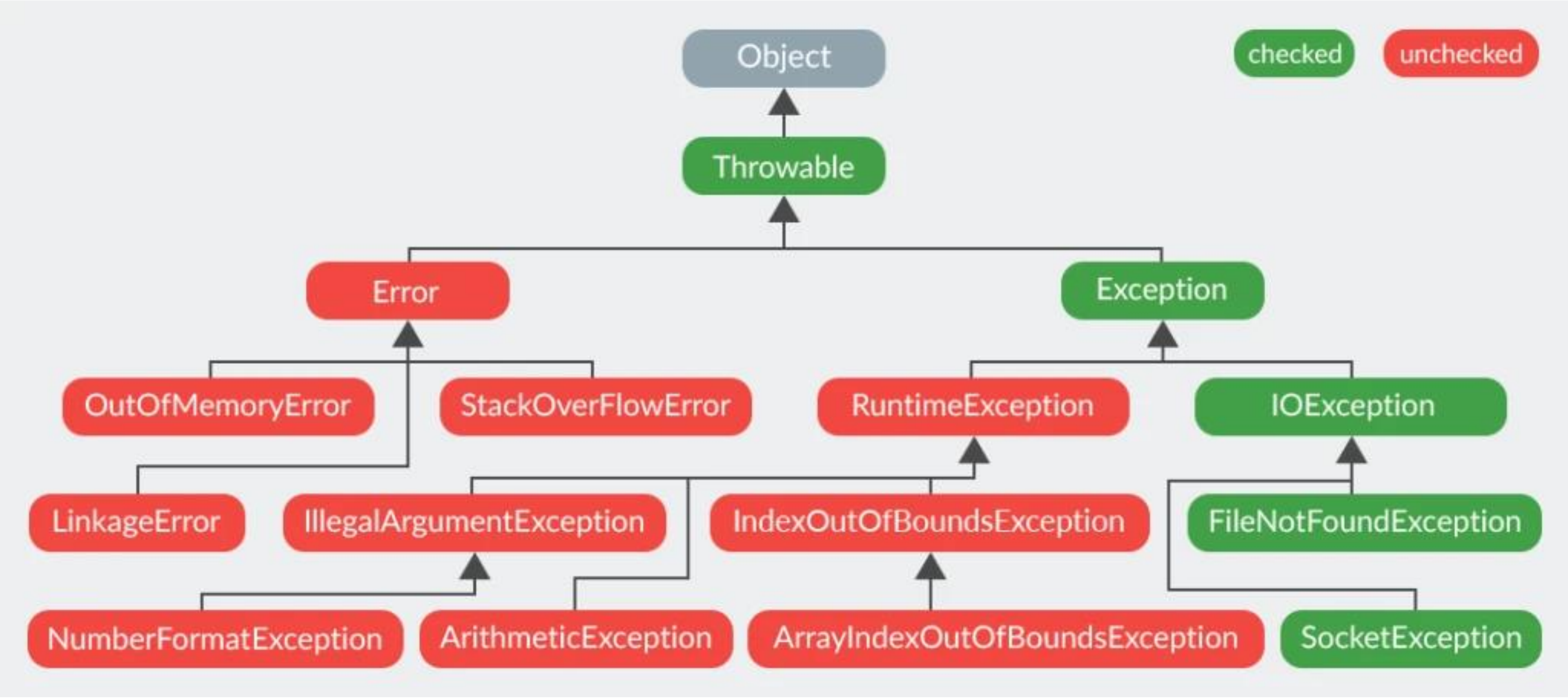
- Overflow and underflow.
  - Underflow: try to pop from empty stack
    - throw RuntimeException if pop from an empty stack
    - all stack implementations have the underflow problem
  - Overflow: try to push in full stack
    - throw RuntimeException if push into full stack
    - only the fixed size array implementation has this overflow problem
    - resizeable array implementation and linked list implementation still have possible an OutOfMemory Error!

- Unchecked exceptions and errors: both RuntimeExceptions and Errors are ***unchecked***:
  - They do not have to be declared with a throws clause in the method signature
  - Client does not have to catch them (try-catch)

Exceptions and Errors in Java

- Exceptions
  - Unexpected events that occurred (unavailable resource, unexpected input, program error,…)
- Errors
  - Errors are typically thrown by JVMs for situations unlikely to be recoverable.
- Exceptions and Errors in Java are Throwable objects that can be thrown by
    - the code or
    - the Java Virtual Machine (run out of memory)
- Exceptions can be caught by a surrounding block of code with try-catch
- Exception can be caught by the method caller’s surrounding block
  - Uncaught exceptions cause Java virtual machine to stop running the program

A Part of the Exception Hierarchy in Java



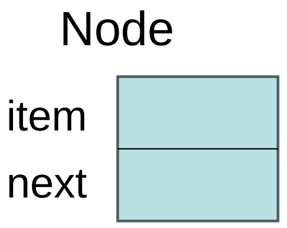
Checked vs Unchecked Exceptions

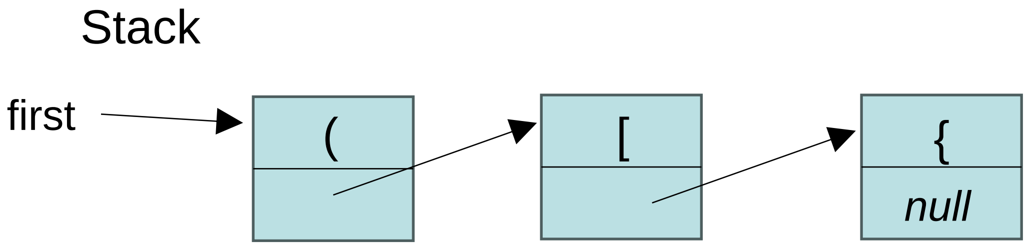
- Checked Exceptions
  - Exceptions that are checked at compile-time
  - Represent recoverable conditions, where the program can handle the exception gracefully, such as: IOException
  - Handling: Must be either caught with a try-catch block or declared using throws in the method signature in order to propagate upwards
- Unchecked Exceptions
  - Exceptions that are not checked at compile-time
  - Typically caused by programming logic errors.
  - All subtypes of RuntimeException are unchecked exceptions. (e.g., NullPointerException, ArrayIndexOutOfBoundsException).
  - Handling: Optional to handle. Can occur during runtime.

Stack Client – Parentheses Matching

```
public class MatchingParentheses {

    public static boolean isMatched(String expression) {
        final String opening = "({["; // opening delimiters
        final String closing = ")}]"; // respective closing delimiters
        StackOfChars stack = new ArrayStackOfChars(10);
        for (char c : expression.toCharArray()) {
            if (opening.indexOf(c) != -1) // this is a left delimiter
                stack.push(c);
            else if (closing.indexOf(c) != -1) { // this is a right delimiter
                if (stack.isEmpty()) // nothing to match with
                    return false;
                if (closing.indexOf(c) != opening.indexOf(stack.pop()))  // mismatched
                    return false;
            }
        }
        return stack.isEmpty(); // were all opening delimiters matched?
    }
//  main …
```

Another Stack Implementation: Linked List





```
class Node {
    private char item;
    private Node next;
}
```

```
public class ListStackOfChars implements
StackOfChars {
    private Node first;     // top of stack

//…
}
```

Class Node must be seen nowhere else outside the ListStackOfChars class

Class Node can be ***nested*** as an ***inner class*** of ListStackOfChars

```
public class ListStackOfChars implements StackOfChars {
    private Node first;    // top of stack

    // helper class - linked list node
    private class Node {
        private char item;
        private Node next;
    }

public ListStackOfChars() {   first = null;    }

public boolean isEmpty() {   return first == null;    }

public void push(char item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

public char pop() {
        if (isEmpty()) throw new RuntimeException("Stack underflow");
        char item = first.item;        // save item to return
        first = first.next;            // delete first node
        return item;                   // return the saved item
    }
}
```

Class Node is a private nested class.
Only class ListStackOfChars can see and use it.
Outside ListStackOfChars, the name Node can be freely reused

Stack Client – Parentheses Matching

```
public class MatchingParentheses {

    public static boolean isMatched(String expression) {
        final String opening = "({["; // opening delimiters
        final String closing = ")}]"; // respective closing delimiters
        StackOfChars stack = new ListStackOfChars();
        for (char c : expression.toCharArray()) {
            if (opening.indexOf(c) != -1) // this is a left delimiter
                stack.push(c);
            else if (closing.indexOf(c) != -1) { // this is a right delimiter
                if (stack.isEmpty()) // nothing to match with
                    return false;
                if (closing.indexOf(c) != opening.indexOf(stack.pop()))  // mismatched
                    return false;
            }
        }
        return stack.isEmpty(); // were all opening delimiters matched?
    }
//  main …
```

Only a minor change in client when switching to another implementation!

Nested classes

- Nested class
  - A class defined within the definition of another class
  - Increase encapsulation
  - The containing class is known as the ***outer class***. The ***nested class (or inner class)*** is formally a member of the outer class, and its fully qualified name is *OuterName.NestedName*. (*example: ListStackOfChars.Node*)
  - the use of nested classes can help reduce name collisions: it is ok to have another class named *NestedName* nested within some other class (or as a self-standing class), even in the same package
  - A nested class can have visibility modifiers (e.g., public, private): whether the nested class definition is accessible beyond the outer class definition.

Nested classes: static vs non-static

- Static nested class
  - Similar to traditional classes
  - Its instance has no association with any specific instance of the outer class
- Non-static nested class (inner class)
  - Can be created from within a non-static method of an outer class
  - Inner class instance is associated with the outer class instance that creates it
  - class Node is an inner class of ListStackOfChars

We need more stacks!

- See examples of various applications that need various stacks

Another application of Stack: Matching tags

- Another kind of matching delimiters problem: the validation of markup languages such as HTML or XML.
  - HTML is the standard format for hyperlinked documents on the Internet
  - XML is an extensible markup language used for a variety of structured data sets
- In an HTML document, portions of text are delimited by ***HTML tags***. A simple ***opening tag*** has the form “`<name>`” and the corresponding ***closing*** tag has the form “`</name>`”.
- Examples: `<body> <h1>…</h1> <h2>…</h2> … <ul>…</ul> </body>`
- Ideally, an HTML document should have matching tags

- Solution: exactly the same as for matching parentheses, but we need a **StackOfStrings**

Another application of Stack: Arithmetic expression evaluation

- Infix form for arithmetic expressions:
  - Binary operators are ***in between*** operands: *operand* ***operator*** *operand*
  - 2+3
  - Need to know precedence rules
  - May use parentheses
  - 4*(3+5) or 4*3+5
- Postfix form for arithmetic expressions:
  - Operator appears ***after*** the operands
  - No precedence rules or parentheses!
  - Infix (4+3)*5 has equivalent postfix: 4 3 + 5 *
  - Infix 4+(3*5) has equivalent postfix: 4 3 5 * +

Another application of Stack: Postfix expression evaluation

- Given a postfix arithmetic expression with binary operators + - * / and integer values as operands, compute its value

- Solution: we need a **StackOfIntegers to store operands**

- Scan the input left to right:
  - If operand: push to stack
  - If operator:
    - pop the stack twice
    - apply operator
    - push result back to stack

Stacks of different types

- StackOfCharacters, StackOfStrings, StackOfIntegers, StackOfPersons, StackOfHouses, …

- **Attempt1: Implement a separate stack class for each type.**
  - **@#$*!**
  - Rewriting code is tedious and error-prone
  - Maintaining cut-and-pasted code is tedious and error-prone

Stack of Objects

- **Stacks of different types – Attempt2: Implement a StackOfObjects**
- Implement a stack with items of type Object.
- Due to Polymorphism, clients can push any types of objects in the stack
- Implementation: just replace char -> Object in previous version ☺

```
public interface StackOfObjects {
    void push(Object item);    // insert a new object  onto stack
    Object pop();   // removes and returns object on top of stack
    boolean isEmpty();  // test if stack is empty
}

public class ArrayStackOfObjects implements StackOfObjects {
    private final Object[] a;  // holds the items
    private int n;       // number of items in stack

    public ArrayStackOfObjects(int capacity) {
        a = new Object[capacity];
        n = 0;
    }
…
```

Stack of Objects Discussion

- Handling null items: In this implementation we allow null items to be inserted
  - Another possible policy: we do not allow null items to be inserted. In this case, in case of underflow may return null instead of exception
- Avoid loitering:
  - Loitering = holding a reference to an object when it is no longer needed

```
public Object pop() {
    if (isEmpty()) throw new RuntimeException("Stack underflow");
    Object item = a[--n];
    a[n] = null;
    return item;
 }
```

this version avoids "loitering":
garbage collector can reclaim memory
only if no outstanding references

Issues with Stack of Objects

- Due to Polymorphism, clients can push any types of objects in the stack
- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects stack = new ArrayStackOfObjects(10);

Integer n = new Integer(17);
String s = new String("abc");
```

In *the same stack*, you can push different types of objects!

```
stack.push(n);   // widening conversion Integer -> Object

Integer x = stack.pop();
```

Compile Error!

```
Integer x = (Integer) stack.pop();  // narrowing conversion  Object -> Integer

stack.push(s);

x = (Integer) stack.pop();
```

Runtime Error!

Casting

- Casting with Objects allows for conversion between classes and subclasses.
- A **widening conversion** occurs when a type T is converted into a “wider” (more general) type U (U is a supertype of T)
  - Example:

```
Object a = new Apple(“red”);  // can do any time !
```

- A **narrowing conversion** occurs when a type T is converted into a “narrower” (more specific) type S
  - a narrowing conversion requires an explicit cast
  - Example:

```
Object x = …
Apple a = x;    // compile error!
Apple a = (Apple) x;   // a runtime error is possible if
                          Object x does not hold an instance of Apple!
```

Abstract Data Types and Algorithms
in Object-Oriented Style

Outline:

- OO and Java Concepts Review
- Stacks revisited OO style
- New concepts: Nested classes, Generics

Parametrized Stack

- **Stacks of different types – Attempt3: use Java Generics**
- Java includes support for writing ***generic*** classes and methods
- The Java generics framework allows us to define a class in terms of a set of ***formal type parameters***
  - The formal type parameters can be used as the declared type for variables, parameters, and return values within the class definition.
  - The formal type parameters are later specified when using the generic class as a type elsewhere in a program
  - ***An instantiation of a generic type where all type arguments are concrete types is a concrete parametrized type***

- **Example: Java collections library extensively use generics**
- Example: Stack implementation offered by Java library: java.util.Stack

Example: Using the generic Stack provided by java.util library

```
import java.util.Stack;

public class JavaUtilStackExample {
    public static void main(String[] args) {
        Stack<Integer> s1 = new Stack<>();
        Stack<String> s2 = new Stack<>();

        // Push elements onto the stacks
        s1.push(1); s1.push(2);  s1.push(3); // autoboxing!
        s2.push("abc"); s2.push("xyz");

        // Pop elements from the stacks
        while (!s1.isEmpty()) {
            System.out.println(s1.pop());
        }
        while (!s2.isEmpty()) {
            System.out.println(s2.pop());
        }
    }
}
```

S1 is a Stack of Integer
S2 is a Stack of Strings

Stack<Integer>, Stack<String> are concrete parametrized types, instantiations of the generic type Stack

Class java.util.Stack extends class java.util.Vector, which implements interface java.util.List

Bloated and poorly-designed API for a stack

Design our own Generic Stack API

```
public interface MyStack <T> {
    void push(T item);    // insert a new object of type T onto stack
    T pop();   // removes and returns object on top of stack
    boolean isEmpty();  // test if stack is empty
}
```

T is the formal type parameter. Classes and interfaces enumerate their formal type parameters in angular brackets after their name

Our Generic Stack - with Linked List

```
public class ListStack<T> implements MyStack<T>{
    private Node first;     // top of stack

    private class Node {
        private T item;
        private Node next;
    }

public ListStack() {     first = null;     }

public boolean isEmpty() {   return first == null;     }

public void push(T item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

public T pop() {
        if (isEmpty()) throw new RuntimeException("Stack underflow");
        T item = first.item;
        first = first.next;
        return item;
    }
}
```

T is the formal type parameter

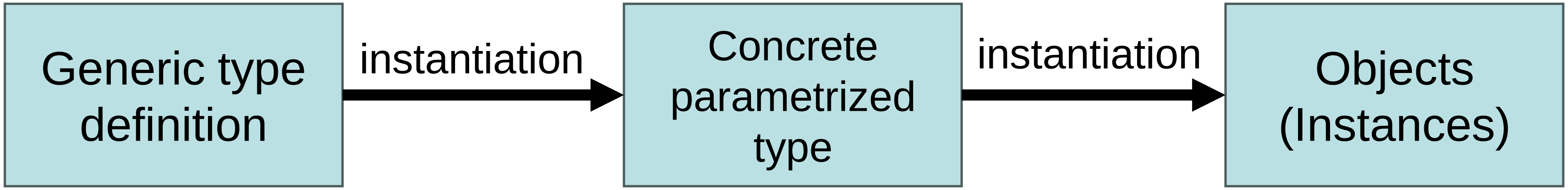
```
public class TestGenericStack {

    public static void main(String[] args) {
        MyStack<Integer> stack1 = new ListStack<Integer>();
        MyStack<String> stack2 = new ListStack<String>();
        MyStack<Person> stack3 = new ListStack<Person>();

        Integer n = new Integer(17);
        String s = new String("abc");
        Person p = new Person(“John“, 24);

        stack1.push(n);
        Integer x = stack1.pop();

        stack2.push(s);
        String s2 = stack2.pop();

        stack3.push(p);
        Person p2 = stack3.pop();


        stack1.push(s);
    }
}
```

Integer, String, Person are **actual type parameters**

Compilation Error!

Generics

- The generics framework allows us to **define generic types** as classes or interfaces with **formal type parameters**.
  - Formal type parameters can then be used as the declared type for variables, parameters, and return values within the class definition.
  - Formal type parameter is a placeholder
- A **concrete parameterized type** is an **instantiation** of a generic type with **actual type arguments**.



Generic type definition

instantiation

Concrete parametrized type

instantiation

Objects (Instances)

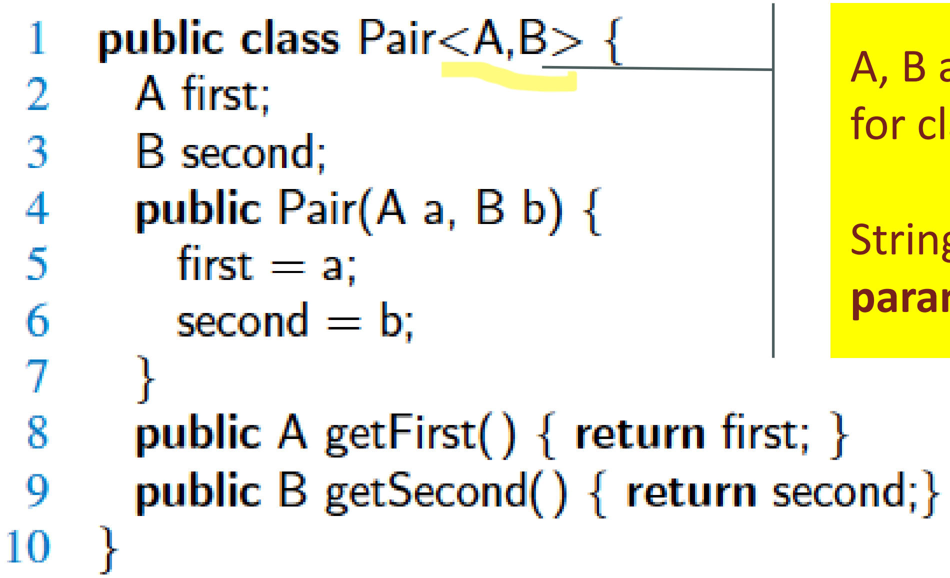
MyStack<T>
ListStack<T>

MyStack<Integer>
ListStack<Integer>

stack1, stack2

Syntax for Generics

- Types can be declared as parametric using generic names (formal parameters):

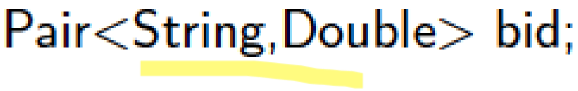


```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {
    first = a;
    second = b;
  }
  public A getFirst( ) { return first; }
  public B getSecond( ) { return second;}
}
```

A, B are **formal type parameters** for class Pair

String, Double are **actual type parameters**

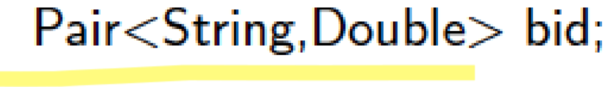
- Generic types are then instantiated using actual type parameters:



```
Pair<String,Double> bid;
```

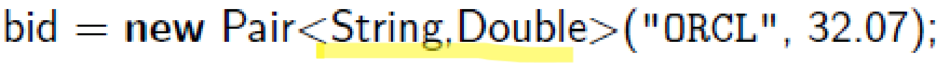
Instantiating type variables

- Generic types are instantiated using actual type parameters:

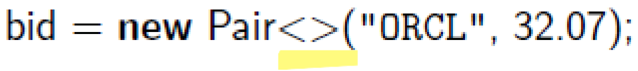


```
Pair<String,Double> bid;
```

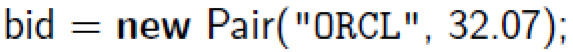
- After this, the variable bid can be instantiated in several ways:

1. Give explicit type: 

   ```
   bid = new Pair<String,Double>("ORCL", 32.07);
   ```
   ✓

2. Rely on type inference 

   ```
   bid = new Pair<>("ORCL", 32.07);
   ```
   ✓

3. Use raw type: 

   ```
   bid = new Pair("ORCL", 32.07);
   ```
   ✗

   – this reverts to the classic style, with Object automatically used for all generic type parameters, and resulting in a compiler warning when assigning to a variable with more specific types.

Instantiating type variables

- ***Different instantiations of the same generic type for different concrete type arguments have no type relationship!***
  - List<Object> is NOT a supertype of List<String>
  - **`List<Object> lo = new List<String>();`** is compile error! ☒
  - But there is such a relationship for arrays:
    - Object[] is a supertype of String[]
- Compatibility between instantiations of the same generic type exist only with ***wildcard instantiations***
  - A wildcard instantiation is a syntactic construct with a " ? " (a question mark) that stands for "all types“
  - **`List<?> lw = new List<String>();`** is ok ☑

Bounded Generic Types

- **A formal type parameter can be restricted by using the extends keyword** followed by a class or interface. In that case, only a type that satisfies the condition is allowed to substitute for the parameter!
- Example:
- Class SportPerson is extended by class Footballer and class RugbyPlayer
- A SportsTeam should hold either Footballers or RugbyPlayers and all members of the team must ***belong to the same sport (the same type)***
- class SportsTeam<**T extends SportPerson**> { … }
- `SportsTeam<Footballers> f1=new SportsTeam<Footballers>();`
- `SportsTeam<Footballers> f2=new SportsTeam<Footballers>();`
- `SportsTeam<RugbyPlayers> r1=new SportsTeam<RugbyPlayers>();`

- Using lists such as `List<SportPerson> f1, f2, r1;` will not do well!

Generics and Arrays

- Java allows the declaration of arrays that store parameterized types, but it **does not fully support the instantiation of arrays involving those types**!
- This limitation manifests in two scenarios:
  - Arrays in Generic Classes: A generic class that declares an array that stores objects of its formal parameter types
    - Example: A generic Stack implemented using an array
  - Arrays Outside Generic Classes: Code outside a generic class may try to declare an array to store instances of the generic class with actual type parameters

The Generic Stack - with Array

```
public class ArrayStack<T> implements MyStack <T> {
    private final T[] a;  // holds the items
    private int n;        // number of items in stack

    public ArrayStack(int capacity) {
        //a = new T[capacity]; NOT ALLOWED!
        a = (T[])new Object[capacity];
        n = 0;
    }

    public boolean isEmpty() {   return n == 0;    }

    private boolean isFull() {  return n == a.length;    }

    public void push(T item) {
        if (isFull()) throw new RuntimeException("Stack overflow");
        a[n++] = item;
    }

    public T pop() {
        if (isEmpty()) throw new RuntimeException("Stack underflow");
        T item = a[--n];
        a[n] = null;
        return item;
    }
}
```

Inside the generic class, we can declare an array with elements of the type T

Generic array creation not allowed in Java!
Here we still have to use the ugly unchecked cast!

Generic Methods

- Not only types can be generic, but methods can be generic, too.
- A method of a non-generic class can have type parameters (a generic method)
- We will encounter such an example next lecture for sorting

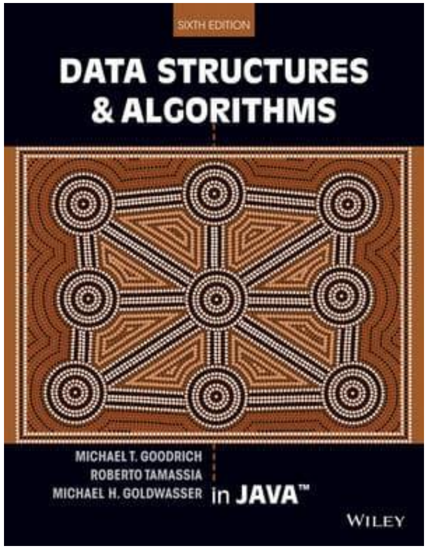
Source Code Examples

- MyStack.java
- ArrayStack.java
- ListStack.java
- TestGenericStack.java

Summary

- Java and OO Concepts review
  - Objects and Classes
  - Classes and Interfaces
  - Exceptions
  - Casts
- Abstract Data Types:
  - Interface – Implementation – Clients
  - Stack revisited OO style
- Algorithms:
  - Some applications using stacks
- New Concepts:
  - Generics

Bibliography



- Sedgewick, chapter 1.3
- Goodrich: chapter 2, chapter 6.1
- Coursera: Algorithms Part I, Module 4 https://www.coursera.org/learn/algorithms-part1/home/module/4