Prep: bring ID, water, lots of paper, pen and this set of notes

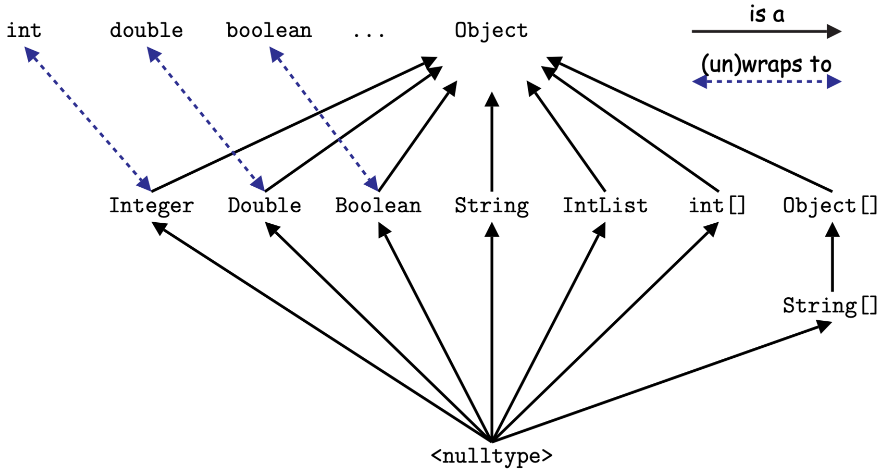
You got this!

Dynamic and Static Types

- A container with static type `T` can only contain dynamic types that are subtypes of `T`.
- A method with return type `T` can only return dynamic types that are subtypes of `T`.
- A method's parameters type `T` defines its static type and can only hold subtypes of `T`.
- If `E[i]` is present, `E` must have an array type.

```
int[] A = new int[2];
Object x = A;
System.out.println(x[0]); // ERROR, x's static type is Object. compiler doesn't know
x is an array
```

- Reference types form a type hierarchy. All types are subtype of themselves.
- `null`'s type is a subtype of all reference type.
- All reference types are subtypes of Object.
- Static type of a casted `(T)E` is `T`.



Variable Selection

- Fields hide inherited fields of the same name. Fields always depend on the **static type**.

```
A b0 = new B();
System.out.println(b0.x); // equals A.x
```

- If you cannot find a variable, go up the hierarchy.
- In a class (say `A`), `this` has static type `A`. (i.e. a variable in a class is that class's variable).
- In a method `int f(B x){}`, parameter `x` has static type `B`.
- `super.super.y` is illegal. Can only get grandparents' variables if parent has helper function.
- `super.y` where `y` does not exist in parent class is also illegal.

Dynamic Method Selection

i.e. `x.f()` and `f()` is an instance method call (Note that `x` can be `this`)

1. Do a compile-time analysis: use static type to judge if will CE. If not, record down the signature of the method (name, parameters, return type).
2. Do a run-time analysis: if dynamic type causes overriding of method, then pick the dynamic type method to replace.

Static Method Selection

i.e. `x.f()` and `f()` is an static method call (Note that `x` can be `this`)

1. Do a compile-time analysis: use static type to judge if will CE. If not, that is your method.
2. If cannot find the static method, go to its parent and find the same static method.

Static methods hide static methods of the same signature.
If `B` extends `A`, it is not allowed for `B` to have a static method of the same name as `A`'s instance method and vice versa.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Table 1: Defining a Method with the Same Signature as a Superclass's Method

Testing (Assert + JUnit)

```
assert N == R.length && N == B.length;
assert x > 0 : "x must be positive, but x = " + x;
```

The Java annotation `@Test` on a method tells the Junit machinery to call that method.

Typical usage	With imports and main function

`assertArrayEquals(expected, actual);`
 	
`assertTrue(<boolean>);`
`i.e. assertTrue(L1.contains(0));`
 	
`assertFalse(<boolean>);`
`i.e. assertFalse(L1.contains(3));`
 	
`assertEquals(<String>, <String>);`
`assertEquals(<Integer>, <Integer>);`
`e.g. assertEquals(2, (int) L1.removeLast());`
 	
`assertNotEquals(<Object>, <Object>);`
	`import org.junit.Test;`
	`import static org.junit.Assert.*;`
	`public class SortTesting {`
	`    private String[] f(String[] input, int L, int U) { //logic }`
	
	`    @Test`
	`    public void emptyTests() {`
	`        String[] x = f(new String[]{}, 0, -1);`
	`        assertArrayEquals("Empty array failed", new String[]{}, x);`
	`    }`
	
	`    public static void main (String... args) {`
	`ucb.junit.textui.runClasses(SortTesting.class);`
	`    }`
	
	`}`

Common Overridden Methods

1. `.toString();`

`toString()` is a function of Object, so it is defined on all objects (no need casting)

```
@Override
public String toString(){
      StringBuffer b = new StringBuffer();
      b.append("[");
      for (IntList L = this; L != null; L = L.tail) b.append(" " + L.head);
      b.append("]");
      return b.toString();
}
```

2. `.compareTo();`

```
// look at Appendix II Comparable
```

3. `.equals();`

`Object`'s `.equals()` method has signature `boolean equals(Object other)` and will only be overridden by a method with the same signature.

```
@Override
public boolean equals(Object o){
      Student other = (Student)o;
      return name.equals(other.name);
}
```

Miscellaneous

```
T[] newItems = (T[]) new Object[capacity]; // generic array declaration
int limit = (int) Math.round(Math.sqrt(x)); // Math.round(), Math.sqrt()
Integer.parseInt(args[0]); // String to Integer
(int) args[0]; // Illegal: "Cannot cast from String to int"
```

Final Checks

- Read descriptions of functions and variables properly. Do **NOT** skim through those!
- Appreciate all methods of an `Object`. Think about why they are declared.
- After finishing compile-time analysis, do **NOT** forget run-time analysis also.
- Java **passes by value** (a copy of the reference is created)

e.g. if you copy a pointer and reassign the pointer, nothing happens (remember practice midterm)

- Non-destructive methods must return something, i.e. cannot be `void`
- Pointers might point to the same thing!
- When constructor is called, the parent constructor is always called first.
- Do not use `str[i]`; use `str.charAt(i);`
- Use `.equals()` for objects instead of `==`
- Default value for `int` is 0; default value for `boolean` is `false`
- Remember the keyword `new` in `throw new IllegalArgumentException();`
- For `IntLists`, did you construct all the links and destroy all the links that you wanted?
- Always check for **edge cases**: empty array, `IntList` pointer pointing to `null` etc.
- Remember to increment your counters (mod them if necessary): indices, `L = L.tail;`

Stuck? Last Resorts

- `System.arraycopy(…) // exploit arraycopy`
- `(x == null)?1:0; // ternary operator all the way`
- `last = last.tail = new IntList(); // multiple assignment`
- `for (int i = 0; i < n; i++); // exploit for loops`
- `if (lists[i] == null) continue; // run on condition/loop`

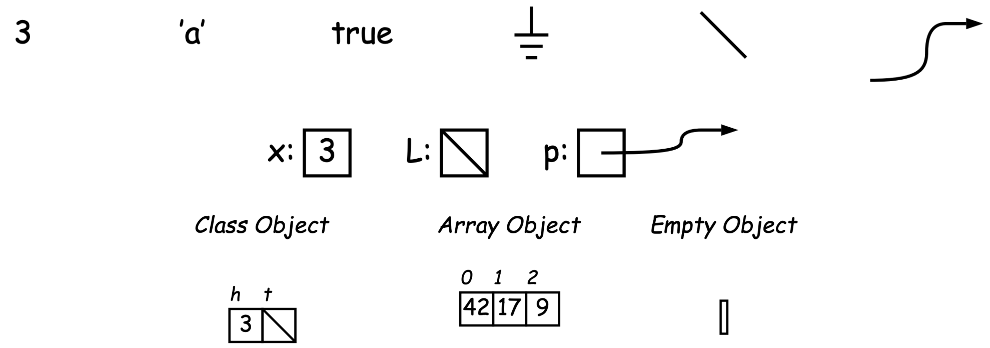
Appendix (Probably not needed, but just in case)

Definitions

- Overloading: multiple method definitions with same name, but different no/type of arguments
- Casting: Tells the compiler how to treat the type of an object **temporarily** at compile-time
- Final: variable's value may not be changed after initialization.
- Can cast from superclass to subclass (downcasting)
- Can cast from subclass to superclass (upcasting, but technically no need to do so)
- If unrelated classes at compile time, compile error "`Cannot cast from <x> to <y>`"
- If unrelated classes or direct subclass at runtime, runtime error

Unrelated Class	Direct Subclass

C is subtype of A; B is a subtype of A; C, B unrelated
A x1 = new C();
B x3 = (B) x1;
	B is subtype of C, which is a subtype of A
	A x1 = new C();
	B x3 = (B) x1;



OOP

- class declaration defines a new type of object
- instance variables are simple containers within the object
- instance methods are like static methods with the invisible **this**.
- method selection picks which method to call.
- Abstract class at the end of array to handle possible errors.

Constructors

- Constructors are special methods used to instantiate new instances; take argument from `new`
- Overloaded constructors are possible.
- When one class extends another, Java guarantees that the parent's constructor is called first. By default, Java calls the default parameter-less constructor `super();`
- All classes have constructors. In the absence of explicit constructor, get the default constructor with no arguments.

Typical Constructor	Default Constructor	Error

`public class A {`
`    public A(int x, int y){}`
`    public A(int x){this(x, 0);`
`// calls first constructor }`
`}`
	`public class A {`
	`    public A() {}`
	`}`
		`class A {`
		`    public A(int x){}`
		`}`
		`class B extends A {`
		`}`

- If no constructor is defined, then the default (empty) constructor is defined for you. However, if one is defined, then no default (empty) constructor is created. Hence, the above code errors "`implicit super constructor A() is undefined for default constructor`" when it implicitly calls super.

Using overriden method	Static variables (class-wide)

`class A {`
`      int f(int x, int y){return res;};`
`}`
 	
`class B {`
`      int f(int x, int y){`
`            int res = super.f(x, y);`
`            // super.super INVALID SYNTAX`
`            return res;`
`      }`
`}`
	`static int _var;`
	`int _val;`
	`static void incr(){ _var = _var + 1; }`
	`void decr(){ _var = _var – 1; }`
	`A x = new A()`
	`System.out.println(A._var); // OK`
	`System.out.println(x._var); // OK`
	`A.incr() // OK`
	`x.incr() // OK`
	`A.decr() // ERROR`
	`x.decr() // ERROR`
	`System.out.println(A._val); // ERROR`
	`System.out.println(x._val); // OK`

Bottom line: An instance of the class can access both instance and static variables and can call both instance and static methods (although editors might complain). When directly accessing from class, can **ONLY** access static variables and use static methods since there is no instance.

Packages

- Packages are a collection of related classes and other packages, sorted by file system
- By default, a class is put into anonymous package. To put it elsewhere, use package declaration at **the start of file.**
- In another package, like `P2`, to use class `C1` in `P1`, must use it like `P1.C1` and not just `C1`. Within the same class, no need declare package.

File Structure	Compile Command and Code

`P1/`
`      C1`
`      C2`
`P2/`
`      C3`
`      C4`
`      C5`
	`$ javac P2/C5.java`
	`$ java P2.C5`
	
	`package P2;`
	`public class C4 extends P1.C2 {`
	`      public static void main(String… args){`
	`            P1.C1 y = new P1.C1();`
	`            C3 z = new C3();`
	`      }`
	`}`

Access Control

- Accessibility is always defined by **static types**. `x.f()` always looks at static type of `x` (enforced by compiler)
- Accessibility depends on how the member's declaration is qualified and where it is being accessed.
- Members (field, method, constructor, nested type) may have any of the four access levels.
- May override a method with one that is at least as permissive an access level. Otherwise, will get the compile error "Cannot reduce the visibility of the inherited method from C1" and "Attempting to assign weaker access privileges".

Public	available **everywhere** (different package, different class)
Protected	available **only within the same package**
Package-Private (default)	same as package private within the package outside the package, available within subtype (say `C2`), but only if accessed from expressions whose static type are subtypes of `C2`.
Private	available only within the **same class**

Inner classes (Non-static nested classes)	Instance of

class A {
public class B {
public void call(int x){
A.this.connectTo();
// A.this refers to the A that created this
}
}
}
 	
A e = new A();
A.B p0 = e.new Account();
A.B p1 = e.new Account();
	if (x instanceof StringReader){
	// logic
	} else if (x instanceof FileReader){
	// logic
	}

Arrays

- Arrays are anonymous, like other structured containers

Declaration	Usage

int[] x = {1, 2, 3};
int[] x = new int[] {1, 2, 3};
String[] x = {“hi”, “hello”, “bello”};
String[] x = new String[] {“hi”, “hello”, “bello”};
 	
int[][] A = new int[3][];
A[0] = new int[] {2, 3, 4};
A[1] = new int[] {2, 3};
A[2] = new int[] {2};
 	
int[][] A = new int[][] {{2, 3, 4}, {2, 3}, {2}};
 	
// defined classes
Dog[] x = new Dog[100];
// generic types
T[] x = (T[]) new Object[100];
	import static java.lang.System.arraycopy;
	arraycopy(arr, 0, result, 0, k);
	arraycopy(src, srcidx, dest, destidx, length);
	
	for (int i = 0; i < a.length; i++) a[i]++;
	
	// List of anything with Object[]
	// remember to cast to access methods
	
	Object[] t = new Object[2];
	t[0] = new IntList(3, null);
	t[1] = "Stuff";
	IntList tList = (IntList) t[0];
	System.out.println(tList.head); // OK
	System.out.println(((IntList) t[0]).head); // OK
	System.out.println(t[0].head); // ERROR

Boxing and Unboxing

Byte	Long	Float	Short	Char	Double	Integer	Boolean
byte	long	float	short	char	double	int	boolean

Inheritance

- All classes extend `java.lang.Object`
- Subtype inherits ALL fields and methods of its direct superclass and passes to subtypes.
- In `B`, you can override an instance method (best practice to put `@Override`)
- You CANNOT override static methods, but you can *hide* them
- All subclasses will have **at least** the methods listed by the superclass.

```
class B extends A { … }
```

Abstract Methods and Classes

- Abstract classes can have instance variables (which are inherited by subtypes).
- Abstract classes can have private methods.
- Instance method can be abstract (no body given, must be supplied in subtype)

- Abstract classes can have constructors (however, it is only used when a non-abstract subtype calls on the constructor of its abstract parent class).

Abstract Class and Usage	Extending an Abstract Class

`public abstract class A {`
`      public abstract void scale();`
`      public abstract void draw();`
`}`
 	
`// X and Y extends A`
`A[] t = { new X(3, 4), new Y(2, 2) };`
	`public class X extends A {`
	`      @Override`
	`      public void scale(){ // logic };`
	`      @Override`
	`      public void draw(){ // logic };`
	`}`
	
	`void drawAll(A[] tA) {`
	`    for (A t : tA) t.draw();`
	`}`

Interface

- Interfaces should only contain static constants and abstract methods (i.e. **no state**)
- Interfaces are automatically abstract, can use in the same way as abstract classes
- You can extend only one class, but implement any number of interfaces (`implement B, C`)
- There is **NO** way to create a default constructor.
- Interface variables are static and final, because they cannot be instantiated in their own right, so the values must be assigned in a static context in which no instance exists.
- Interfaces cannot have private methods.

```
public interface IntUnaryFunction { int apply(int x); }

class Abs implements IntUnaryFunction {
      public int apply(int x) { return Math.abs(x); }
}

IntList map(IntUnaryFunction proc, IntList items) {
      if (items == null) return null;
      else return new IntList( proc.apply(items.head), map(proc, items.tail));
}

r = map(new Abs(), lst);
r = map(new IntUnaryFunction(){public int apply(int x){return Math.abs(x);} }, lst);
r = map((int x) -> Math.abs(x), lst);
r = map((x) -> Math.abs(x), lst);
r = map(Math::abs, lst);
```

Default Methods

```
public interface Drawable {
      void scale(double xsize, double ysize);
      void draw();
      default void scale(double size) { scale(size, size); }
}
```

Integers

Type	Bits	Signed	Type	Bits	Signed
Byte	8	Yes	Int	32	Yes
Short	16	Yes	Long	64	Yes
Char	16	No			

- A signed N bit goes from -2^{N-1} to 2^{N-1}
- An unsigned N bit goes from 0 to $2^N - 1$

- Java wraps around (performs operation, then take value that is in the range)
- Java silently convert data type if it makes sense and no information is lost from value.
- Arithmetic operations +, × promote (implicitly convert) operands when needed
    - If any operand is long, promote both to long
    - Else promote both to int

```
byte a = 0;
a = a + 3; // Illegal "Type mismatch: cannot convert from int to byte"
a += 3; // works
```

Bitwise Operations

Shift Left (<<)	Shift Right (>>)
$x \ll n = x \cdot 2^n$	$x \gg n = \lfloor x/2^n \rfloor$
Least Significant Bit	Carrying

x &= -x; // signed
x &= ~x + 1; // unsigned
	z = x ^ y; // no carry
	x = (x & y) << 1; // carry
	y = z; // repeat log N times

Appendix II: Regex

Anchors		Quantifiers	
^The	matches any string that starts with The	abc*	ab with 0 or more c
end$	matches any string that ends with end	abc+	ab with 1 or more c
^The end$	matches the exact string	abc?	ab with 0 or 1 c
roar	matches any with roar	abc{2}	ab followed by 2 c
Character Classes		abc{2, }	ab followed by 2 or more c
\d (negated \D)	single character that is a digit	abc{2, 5}	ab followed by 2 up to 5 c
\w (negated \W)	matches a word character (alphanumeric or underscore)	a(bc)*	a with 0 or more copies of bc
\s (negated \S)	matches a whitespace character (including tabs and line breaks)	a(bc){2, 5}	a with between 2 to 5 copies of bc
.	matches any characters		
Special Characters		Or Operator	
\^.[$()|*+?{\	Escape using \	a(b|c) a[bc]	Matches a followed by b or c

Appendix III: Data Structures

(Just the same as 6 years ago, eh?)

Note: interfaces **extend** other interfaces, while classes **implement** interfaces.

String

```
String x = “Bellow!”;
System.out.println(x.length());
System.out.println(x.toUpperCase());

System.out.println(x.toLowerCase());

int t = x.indexOf(“low”); // t = 3

String y = “Bellow!”;
x.equals(y); // true

y.charAt(0); // ‘B’

// sidx inclusive, eidx exclusive
System.out.println(s.substring(sidx));
// sidx inclusive, eidx = s.length()

x.compareTo(y); // lexigraphic order
```

Exception

The object thrown by throw must be a subtype of `Throwable`
`throw` method causes each active method call to terminate abruptly
try, catch block will allow us to go on with life

```
throw new SomeException();
throw new SomeException(“optional description”);

try {
    // logic
} catch (AException e){
    System.out.println(e.getMessage());
} catch (BException e){
    System.out.println(“Why?!”);
}

// catching multiple exceptions
try {
    // logic
}                                              catch
(IllegalArgumentException|IllegalStateException ex){
    // logic
}
```

List (Interface, extends Collection)

```
import java.util.List;

add (E e); // add element to end of list

add(int index, E e); // add element at position

remove(Object o); // remove first occurrence of o

remove(int index); // remove element at index

contains(Object o); // check if list contains o

get(int index); // get element at index

isEmpty(); // check if list is empty
```

ArrayList

```
import java.util.ArrayList;

ArrayList<String> cars = new ArrayList<String>();
cars.add(“BMW”);
String c = cars.get(0); // c = “BMW”
cars.set(0, “Opel”);
cars.remove(0);
cars.clear();
System.out.println(cars.size());

for (int i = 0; i < cars.size(); i++){
    System.out.println(cars.get(i));
}
for (String i : cars) System.out.println(i);

Iterator<String> iter = cars.iterator();
while (iter.hasNext()){
    System.out.print(iter.next() + “ “);
}
```

Set (Interface, extends Collection)

```
add (E e); // add e to set

remove(Object o); // removes o from set

contains(Object o); // check if set contains o

isEmpty(); // check if set is empty
```

HashSet

```
import java.util.Hashset;

HashSet<String> cars = new HashSet<String>();
cars.add(“Volvo”);
System.out.println(cars.contains(“Mazda”)); // false
cars.remove(“Volvo”);
cars.clear();
System.out.println(cars.size()); // 0
for (String i : cars){
System.out.println(i);
}

HashSet<Integer> numbers = new HashSet<Integer>();
// use wrapper class Integer, Double, Boolean
```

Iterator (Interface)

```
import java.util.Iterator;

boolean  hasNext();  //  true  if  iteration  has  more
elements

E next(); // returns next element in the iteration;
if none left, throws NoSuchElementException

default void remove();
default  void  forEachRemaining(Consumer<?  super  E>
action);
```

```
for    (Iterator<String>    i    =    L.iterator();
i.hasNext();){
    String value = i.next();
    System.out.print(value + “ “);
}

Iterator<String> i = L.iterator();
while (i.hasNext()){
    String value = i.next();
    System.out.print(value + “ “);
}
```

Iterable

```
import java.util.Iterator;
public class A implements List<Integer>,
Iterable<Integer>{
    private class AIt implements Iterator<Integer>{
        public Integer next(){ // logic }
        public boolean hasNext(){ // logic }
    }
    public Iterator<Integer> iterator(){
        return new AIt();
    }
}
```

```
// A class that implements Iterable interface must
provide an iterator method that returns an iterator.

A list = new A();
Iterator<Integer> iter = list.iterator();
int m = iter.next();

for (Integer x: list){
    System.out.println(x);
}
```

Reader

```
import java.io.Reader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class ReaderA{
    public static void main(String[] args) throws
FileNotFoundException, IOException{
        Reader r = new FileReader(“file.txt”);
        int c = r.read();
        while (c != -1){
            System.out.print((char)c);
            c = r.read();
        }
        System.out.println();
    }
}
```

Comparator

```
class RatingCompare implements Comparator<Movie>{
    // negative if m1 less than m2
    public int compare(Movie m1, Movie m2){
        return m1.getRating() – m2.getRating();
    }
}

public class XComp implements Comparator<String>{
    public int compare(String x, String y){
        return count(x, ‘x’) – count(y, ‘x’);
    }
}
```

Comparable

Returns < 0, == 0, > 0 if `this` is less than, equal to or greater than obj

```
public interface Comparable{
    int compareTo(Object obj);
}
public interface Comparable<T>{
    int compareTo(T x);
}

class Movie implements Comparable<Movie>{
    // negative if this is less than m
    public int compareTo(Movie m){
        return this.year – m.year;
    }
}

public static Comparable max(Comparable[] A){
    if (A.length == 0) return null;
    Comparable res; res = A[0];
    for (int i = 1; i < A.length; i += 1){
        if (res.compareTo(A[i]) < 0) res = A[i];
    }
    return res;
}

class A implements Comparable {
    @Override
    public int compareTo(Object obj){
        A x = (A) obj; // don’t forget to CAST
        return cnt – x.cnt;
    }
}

// eliminates the need for casting
class T implements Comparable<T>{
    @Override
    public int compareTo(T obj){
        return cnt – x.cnt;
    }
}
```

Consumer (Interface, equivalent to lambda)

```
void accept(T t);

default Consumer<T> andThen(Consumer<? super T>
after)

import java.util.function.Consumer;
Consumer<Integer> lam = a -> System.out.println(a);
lam.accept(10); // prints 10

Consumer<List<Integer> > modify = list -> {
    for (int i = 0; i < list.size(); i++){
        list.set(i, 2 * list.get(i));
    }
};
List<Integer> list = new ArrayList<Integer>();
list.add(2);
list.add(3);
modify.accept(list);
// list is now {4, 2, 6}

class Lambda {
    static void f(List<String> L, Consumer<String>
action){
         for (String x : L) action.accept(x);
    }
}
class Printer1 implements Consumer<String> {
    // if Consumer instead, override accept(Object y)
    public void accept(String y){
        System.out.println(y);
    }
}
Lambda.f(L, new Printer1());
Lambda.f(L, (y) -> System.out.println(y + y));
Lambda.f(L, (y) -> {
    System.out.print(“Multi-line”);
    System.out.println(y);
});
```

Pattern

```
import java.util.regex.Pattern

String x = “(ab)”);
boolean c = x.matches(“[(].*[)]”);

Scanner _input;
boolean d = _input.hasNext(“[*]”);
```

Scanner

```
import java.util.Scanner;
public class ScannerA{
    Scanner s = new Scanner(System.in);
    String input = s.nextLine(); // get a line
    int x = s.nextInt();
    }
}
```