CS2030S

Java Types

Primitive vs Wrapper Class

Each Java primitive type has a corresponding wrapper class.

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`boolean`	`Boolean`
`char`	`Char`
`float`	`Float`
`double`	`Double`

In particular, `Integer` extends `Number`.

Implicit vs Explicit Type Casting

For primitive data types:

`byte < short < char < int < long < float < double`

Any type conversion from left to right is implicit casting (done by Java), while right to left is explicit casting (must be typecasted manually).

For example,

```
// this can compile
// (known as widening conversion)
byte b1 = 127;
short s1 = b1;

// this can compile only because of the typecast
// note the loss of precision
// (known as narrowing conversion)
short s2 = 127;
byte b2 = (byte) s2;
```

Compile-time vs runtime types

Compile-time type

- The type in which the variable is declared
- Restricts the methods it can call during compilation

Runtime type

- The type of the object that the variable is pointing to
- Determines the actual method called during runtime

An example

```
Integer i = 0;
Object o = (Number) i;
```

Variable	Compile-time type	Runtime type
i	Integer	Integer
o	Object	Integer

Also, the code will compile because a `Number` is an `Object`, so the type conversion is valid.

Another example

```
String s = "abc";
Object o = s;
String t = (String) o; // the typecast is required for the program to compile
```

Variable	Compile-time type	Runtime type
o	Object	String
t	String	String

Type Erasure

- `Parent<S>` and `Parent<? extends S>` are considered two different types at compile time, despite both being erased to `Parent` after erasure.
- If we have `public int sort(Array<String> arrayString)` and `public int sort(Array<Integer> arrayInteger)` as two methods of the same class, then type erasure will cause the two methods to have the same method signature. Thus the code will not compile.
- Unbounded type parameters, e.g. `S` gets replaced with `Object` after erasure.
- Bounded type parameters, e.g. `E extends Comparable<E>` gets replaced with the first bound class, `Comparable` in this case.

Variance of Types

Let `<:` denote a sub-type (substitutability) relationship.

- Java arrays are **covariant**,
  - `C <: S` implies `C[] <: S[]`

- Java generics is **invariant**,
  - `C <: S` does not imply `ImList<C> <: ImList<S>` and
  - `C <: S` does not imply `ImList<S> <: ImList<C>`
- Parameterized types are **covariant**,
  - `ArrayList <: List` implies `ArrayList<C> <: List<C>`
- `? extends` is **covariant**,
  - `C <: B` implies `ImList<C> <: ImList<? extends B>`
- `? super` is **contravariant**
  - `B <: F` implies `ImList<F> <: ImList<? super B>`

Java (Misc)

Qualified `this`

```
class Envelope {
    void x() {
        System.out.println("Hello");
    }
    class Enclosure {
        void x() {
            // Qualified this, refers to the x() above
            // with println
            Envelope.this.x();
        }
    }
}
```

Accessibility of modifiers

	Class	Package	Subclass (same package)	Subclass (diff package)	World
public	x	x	x	x	x
protected	x	x	x	x	
no modifier	x	x	x		
private	x				

Static (Early) vs Dynamic (Late) Binding

The binding of overloaded methods is static, and the binding of overriden methods is dynamic. - Since binding of overloaded methods is static, then the `.equals()` method is decided by compile-time type.

In general, calls to `final`, `private` or `static` methods will have static binding. The `final` keyword explicitly prevents overriding.

Example

Because `Human.walk()` and `Boy.walk()` are both static methods, they cannot be overriden, thus the comjpile is able to determine which method will be called at compile time. Thus static (early) binding occurs.

On the other hand, `Human.run()` and `Boy.run()` are both not static. Thus, which method to use is determined at runtime, resulting in dynamic (late) binding.

```
class Human {
   public static void walk() {
       System.out.println("Human walks");
   }
   public void run() {
       System.out.println("Human runs");
   }
}
class Boy extends Human {
   public static void walk(){
       System.out.println("Boy walks");
   }
   public void run() {
       System.out.println("Boy runs");
   }
}
class Main {
   public static void main(String args[]) {
       Human obj = new Boy();
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
       obj.run();
       obj2.run();
   }
}
```

Output:

```
Human walks
Human walks
Boy runs
Human runs
```

Stack, Heap, Metaspace

Stack, Heap

- When an object is created, it is stored in the Heap, and Stack contains the reference to the object.

- Whenever a method is invoked, a block is created in the stack memory, for the method to hold primitive values and references to objects.

Metaspace

- Metaspace contains metadata about the application the JVM is running.
- e.g. class definitions, method definitions, `static` variables, other information about program.

Exceptions

If catching multiple exceptions, make sure that the `catch` blocks are ordered by most specific exceptions first.

Checked vs Unchecked

Checked exceptions represent errors outside the control of the program. Java verifies checked exceptions at compile-time. We should wrap it in a try-catch block.

Unchecked exceptions reflect some error in the program logic. For example, division by 0 will throw `ArithmeticException`. All exceptions that inherit from `RuntimeException` are unchecked exceptions. `ArithmeticException` is one such example. We don't need to wrap it in a try-catch block.

In general, a checked exception is something that the client can rectify. For example, `IncorrectFileNameException` for invalid file name input. On the other hand, if the file name is a null pointer, then it reflects errors in the code, and we should throw an unchecked exception.

Overloading

If there are two methods in a class that have the same name, but they have different method signatures (after type erasure), we can do method overloading.

Ways to do it

1. Number of parameters

```
add(int, int)
add(int, int, int)
```

2. Type of parameters

```
add(int, int)
add(int, float)
```

3. Sequence of type of parameters

```
add(float, int)
add(int, float)
```

Note that the return type is not included in the above examples. This is because the return type is irrelevant to overloading. The following example does not compile:

```
int add(int, int)
float add(int, int)
```

Purpose

Method overloading is useful when we have same functionality across methods, but need to take in different types of input. For example, `int plusMethodInt(int x, int y)` and `double plusMethodDouble(double x, double y)` are two methods with the same functionality. While they do work, having a `int plusMethod(int x, int y)` and `double plusMethod(double x, double y)` is better, because they share an identical method name, so it is unambiguous that they represent very similar functionality.

Comparator and Comparable

Initialization:

```
// list is [1,2,3]
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
```

Method 1: Using `Comparator`, and defining a class

```
class DescendingComp implements Comparator<Integer> {
    @Override
    public int compare (Integer i, Integer j) {
        // return a negative integer if i should come before j
        // return a positive integer if j should come before i
        // return 0 if inconclusive
        return j - i;
    }
}

// result is [3,2,1]
list.sort(new DescendingComp())
```

Method 2: Using `Comparator`, with anonymous inner class

```
// result is [3,2,1]
list.sort(new Comparator<Integer>() {
    @Override
    public int compare (Integer i, Integer j) {
        return j - i;
```

```
    }
})
```

Method 3: Using lambdas

```
// result is [3,2,1]
list.sort((Integer i, Integer j) -> { return j - i; });

// result is [3,2,1]
list.sort((i, j) -> j - i);
```

Misc Stuff

- `@Override` is just an annotation, thus it is optional
- A `Child` that does not throw an exception can substitute a `Parent` that throws an exception. LSP is not violated.
- Given a variable `T t`, you can assign it a value that has a type that is a subtype of `T`.
- When an object is created using `new`, a reference to the instantiated object is returned.

OOP Principles

SOLID

Single Responsibility Principle

> A class should only have one reason to change.

For example, a `Circle` class is in charge of dealing with `Circle`-related computations. It should not have `void print()` method that directly prints to the console, because this means we are now giving it the responsibility to choose where output is redirected to. Instead, we should use a method like `String toString()`.

Open-Closed Principle

> A class should be open to extension but closed to modification.

Consider a `CalculateArea` class that computes the sum of areas of an array of `Shape`. We might do the following if we are only dealing with `Rectangle`:

```
for (Shape S : shapes) {
    area += s.width * s.height;
}
```

But if we were to add a `Circle` class, this code would not work, and this is a violation of the Open-Closed Principle.

Instead, we should have done this:

```
for (Shape S : shapes) {
    area += S.getArea();
}
```

so that we can add new subtypes of `Shape` and implement their `getArea`, without having to change the implementation in `CalculateArea`.

Liskov Substitution Principle

> If S is a subclass of T, then an object of type T can be replaced by that of type S without changing the desirable property of the program.

For example, if `FilledCircle extends Circle`, then anywhere we expect a `Circle`, we can replace it with a `FilledCircle`.

Interface Segregation Principle

> A client should not be forced to depend on methods it does not use.

1. Clients should not implement methods that they can't.
2. Clients should not know of methods they don't need.

For example, if we have an abstract class `Bird` like this:

```
abstract class Bird {
    abstract void fly();
    abstract String getName();
}
```

This violates ISP because not all subclasses of `Bird` will want to implement the `fly()` method (not all birds fly). Instead we should do something like this:

```
interface FlyingAnimal {
    abstract void fly();
  }

abstract class Bird {
    abstract String getName();
}
```

Dependency Inversion Principle

> High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

For example, consider this low-level `LightBulb` class:

```
public class LightBulb {
    public void turnOn() {
```

```
        System.out.println("LightBulb turned on!");
    }
    public void turnOff() {
        System.out.println("LightBulb turned off!");
    }
}
```

Now, we have a high-level ElectricPowerSwitch class that is directly dependent on the low-level LightBulb class:

```
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```

The `LightBulb` class is "hardcoded" into the `ElectricPowerSwitch` class. This violates the Dependency Inversion Principle, because an `ElectricPowerSwitch` should work for any electrical appliance, not just a `LightBulb`.

We implement the following interfaces to rectify this issue:

```
public interface Switch {
    boolean isOn();
    void press();
}

public interface Switchable {
    void turnOn();
    void turnOff();
}
```

```
public class ElectricPowerSwitch implements Switch {
  // code here ...
}

public class LightBulb implements Switchable {
  // code here ...
}
```

Tell-Don't-Ask

> Tell an object what to do, don't ask an object for data

Abstraction

Data abstraction

```
// V1
// User has to set x and y separately, accessing two variables
double x = 1.0;
double y = 1.0;

// V2
// User
double[] point = new double[2];
point[0] = 1.0;
point[1] = 1.0;

// V3
Point point = new Point(1.0, 1.0);
```

Define a `Point` class comprising two `double` value properties. Any user of the `Point` class need not know the exact details of how the point is represented, and the implementer of `Point` is free to change the representation without affecting the client.

Functional Abstraction

Example of functional abstraction (not exactly in an OOP way)

```
double distanceBetween(Point p, Point q) {
  double dx = p.x - q.x;
  double dy = p.y - q.y;
  double dist = Math.sqrt(dx * dx + dy * dy);
  return dist;
}
```

Encapsulation

Two types: **packaging**, and **information hiding**

Packaging

Related data or variable can be packaged in a self-contained unit. This can be done by using Classes to package low level data. As programmers, we need to be able to store respectively data in their related object or entity.

Information hiding

Consider a class `Point`. Consider also a class `Circle` with properties `Point centre` and `double radius`.

Assume we want to create a method for the `Circle` class called `boolean contains(Point p)`. Mathematically, if distance from centre to `p` is less than radius, the point is contained in the circle. In this case, we should create a method in `Point` that helps compute the distance, instead of computing based on exposed properties of `Point` inside `Circle`.