CS2030 Notes

Typecasting

1. Lossy conversion not permitted unless you force a typecast.
2. Some types (eg int and boolean) are just not compatible and cannot even be forced typecasted.
3. You can directly cast a subclass (PaintedCircle) to its Superclass (Circle).
3.2 Circle c = new PaintedCircle(); OK
4. You cannot cast (not even force cast) a superclass instance to its subclass (exception / different case from 6)
5. Interfaces are abstract classes, even if empty, cannot be instantiated with new.
5. NOT OK to assign interface instance to a class that implements it (without typecasting)
5. OK to instantiate / assign a class that implements the interface, to an interface reference.
5. A a = new A(); B b = new B(); //Where B extends A
A c = b; //OK, no need force cast.
6. tl;dr i = j --> If i is subclass of j (i builds on j), ok, no need force cast.
If i superclass of j, if j points to a variable that can be cast into I, then ok. But must force cast, if not error.
7. Watch for typo.
7. Summary: Can up-cast without force cast, checked at compile time. Down-cast requires force cast, checked at run time.
8. **TOTALLY unrelated** classes cannot cast to each other even if forced. Compile time check.

```
Integer a = 1;
String b = (String)a; // compile error: incompatible types
String b = (String)(Object)a; // runtime error: ClassCastException
```

[Object is related to String]

Inheritance/Method Overriding

9. Methods are inherited by subclass when not 'overridden' / private.
10. For every object O, after inheritence, when you override, it will override up the chain of inheritence.
Even if you cast this object to its superclass, the overriden method is still overriden.
Of course, init-ing an object as its parent class doesn't override.
11. Still can call super.f() if subclass overrides it. Will call super's actual f(). Can be called from another subclass method too. Pretty obvious considering a memory table.
12. Pretty obvious infinite loop there.
13. Cannot override with the same signature but different return types.
14. Same name, different signature is not "override". Treated as two totally different functions.
15. Nothing too special here. As per (10).
16. !!Private methods are automatically final and hidden from subclasses. They will NOT be overriden. Simply hidden.
16. Remember you cannot access f() because it is private in A.
16.2 (Converse) Also, cannot override public method and then make private. Can only "relax" the access privileges. Compile error: "attempting to assign weaker access privileges; was public"
18. Cannot override a static method by declaring non-static. Similar vice-versa.
19. static methods are, once again, hidden and not overridden. a.f() accesses static method declared in A.
20. Nothing too special. Different method signature.
21. Similar. But same method signature and hence does not compile. Different naming of parameters does not constitute different method signature, only the type matters.
22. Similar. Same method signature. Doesn't matter, in the same class, whether private or public. Refer to 16.2 too
23. Similar. Different return type doesn't affect method signature.
24. OK. Different method signature. (Eg: String a, Integer b vs Integer a, String b)
25. Cannot. Similar method signature. What's worse: one throws and the other doesn't!
25.1. 'throws' are not included in method signature as it cannot be determined which function is it when invoking it.

26. ‘x’ in super class is private, cannot access.
26.1. If ‘x’ in super class is public / protected, sub classes can access it.
26.2 This is allowed,

```
class A {
   protected int x = 2;
 }
 class B extends A{
   private int x = 3;
 }
 class C extends B {
   public void f() {
     System.out.println(x);          //implies super.x which is private, compile time error
   }
 }
```

27. Similar. Super was implied in 26.
28. Similar. Protected is OK.
29. Since x is defined, b does not attempt to look up x in A. Can also ‘extend’/’widen’ scope. No errors regarding “weaker access privileges”.
30. Similar. Protected is accessible. super.x still refers to the parent class’ x.
26-30: If B also defines x, then x is 2 separate matters in A and B. Modifying x in A methods does not affect x in B, vice versa. Variables are not "inherited". If it doesn't exist, the class simply looks for it in the superclass.

Methods can be overriden because they are essentially pointers to functions.
Inheriting from a class (and adding your own methods) builds your methods on top of the pre-existing class.
If you choose to override, you will override the method for your class, not erasing the pre-existing one.
Refer to lecture 3 notes for picture.

Exceptions
31. Somehow java is really smart – it detects unreachable code after “throw new..” and prevents compilation
32. Expected. Compiles.
33. Note that declaring to throw IllegalArgumentException =/= throw Exception(); Still must declare! Exception is also not an unchecked exception, so when thrown, must be checked and caught! Can declare that it throws IllegalArgumentException (a RTException) and still no need to catch because it’s a runtime exception. But declare Exception then must catch.
34. Yup. Can declare to throw its superclass and catch it accordingly. Remember! Must catch checked exceptions! Unchecked ones are subclass of RuntimeException which is a subclass of Exception.
35. Expected. The exception caught is a different one, so it’s essentially not caught. Declared that it throws Exception, so main needs to try catch. If never declare throws exception, main need not try catch (and just crash).
36. Expected caught exception. Then main’s catch doesn’t trigger.
37. Surprisingly, once you catch an exception, you cannot try to catch it again. Doing so results in a compile error! (Java does this sequentially)
38. Also, importantly, each exception will only be caught once. If it’s caught in AIOOB, it won’t be caught again in the generic version.

Auto Boxing and Unboxing

39. Remember auto boxing and unboxing.
40. Can. Auto unboxing.
41. Cannot convert Integer to Double even with a forced typecast. Only int to double.
42. Able to unbox and convert.
43. Refer to (1). Loss of precision = cannot compile unless you force downcast.
44. Even if you forcecast 5 to an int, it will be casted to a double. However, since you casted 2.5 to a double, there will be loss if precision = cannot compile to int.
45. Expected downcast.
46. int value also cannot be converted into a Double directly. Need manual cast to double. Vice versa, double cannot convert to Integer
47. Works as expected. Refer to 46.
48. Integer can be converted to double or int. But double cannot be converted to Integer???
49. Cannot convert 5 to Double, as expected.

v To From ->	**int**	**Integer**	**double**	**Double**	**char**	**Character**
int	-	Auto Unbox	Force Typecast	Cannot	Can	Can (Auto unbox + magic)
Integer	Auto Box	-	Cannot	Cannot	Cannot	Cannot
double	Can	Can (Auto unbox + magic)	-	Auto unbox	Can	Can (Auto unbox + magic)
Double	Cannot	Cannot	Auto box	-	Cannot	Cannot
char	Force Typecast	Cannot	Force Typecast	Cannot	-	Auto Unbox
Character	Cannot	Cannot	Cannot	Cannot	Auto box	-

Generics and Collections

50. Works as expected, Java will auto use the type declared on the left hand side. Also, auto box.
51. 1,2,3,4,5 as expected. Also, auto unbox.
52. Inverse sort as expected (refer to API docs, <0 = smaller, 0 = arbitrary, >0 = larger). Note that -ve is applied last, flipping results.
53. HashSet sorts in order by hash value. No guarantee it'll be 1,2,4,5 if not integer. Also note no duplicate.
54. Remember map.put() api replaces old value with new value.
55. Expected.
56. Sequence is ordered by the HashMap. For integers it's probably by int value.
57. Sequence is ordered by the HashMap. Using hash code, so might not be sorted in keys order.

Others

Defining a Method with the Same Signature as a Superclass's Method		
	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Primitives default to equivalent of 0. All others are “reference types”, including String and **array**, default to null.

```
“-“ + 2 + 3 = -23
3 + 2 + “-“ = 5-
1 +‘0’= 1+48= 49
‘0’+1 = 49
1+“0”=“10”
“0”+1= “01”

double a = 2 * 3.0 → a = 6.0 (Double) OK.
int a = 2 * 3.0 → Compile Error lossy conversion from double to int.

class M {
  public static double f(double a, double b) {
    return a * b * 2;
  }
}
M.f(4, 5.0) → actually returns 40, not error!
```

Reference Codes

```
import java.lang.*;
import java.util.*;
class EventComparator implements Comparator<Event> {
  public int compare(Event e1, Event e2) {
    return e1.isEarlierThan(e2);
  }
}

PriorityQueue<Event> events = new PriorityQueue<Event>(new Comparator());

OR

public class Event implements Comparable<Event> {
  …
  @Override
  public int compareTo(Event e2) {
  }
}

PriorityQueue<Event> events = new PriorityQueue<Event>();
```