CS2030 Cheatsheet(s)

Inheritance

Classic Inheritance

- A subclass inherits all of the public and protected members of its parent
- If the subclass is in the same package as its parent, it also inherits the package-private members of the parent (this is usually the case in single-file declaration)
- The `super.field` keyword can access *overriden and hiddent* superclass fields.
- The constuctor of a subclass can call on the constructor of a superclass. If it does not explicitly do so, the Java compiler will automatically insert a call to the no-argument constructor of the superclass.
- A typical scenario where inheritance is needed is when a subclass does the same things as the superclass and some *additional* functionality.

Access modifiers

= The following table, summarizes the access modifiers:

Access Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Polymorphism (Interface & Late binding)

Consider an interface `I` and a class that implements it, `A`. `I i = new A(); i.f();`

- During *compile* time, Java checks if an object of type `A` can be assigned to a variable of type `I`. In this case, it can.
- During *run time*, Java looks at the object in the heap of type `A`, determines its class and right implementation of `f()`.

Overriding methods:

- When a method is called, we look at its *method signature*. This is i) the method's name and ii) the number, order, and type of the arguments. **return type is not part of signature**.
- Methods with different signature can coexist in a class.
- A method is overriden by a subclass when the subclass has a method with the same signature.

Method tables

- When `Circle` extends `Object`, its method table contains a copy of `Object`'s method table.
- If a method in `Circle` overrides one in `Object`, the relevant pointer in `Circle`'s method table (in the part that was duplicated from `Object`) is changed to point to the new method body.

Stack and Heap diagram

Stack on the right, Heap on the left.
Stack:

- Box with the name outside
- Arrow outwards to the reference on the Heap
- Add a box to encapsulate the variables within a function call frame (ensure parameter names, *not* argument names)
- Stack frames on top of one another

Heap:

- Class name on top
- `<field name>: <value>`

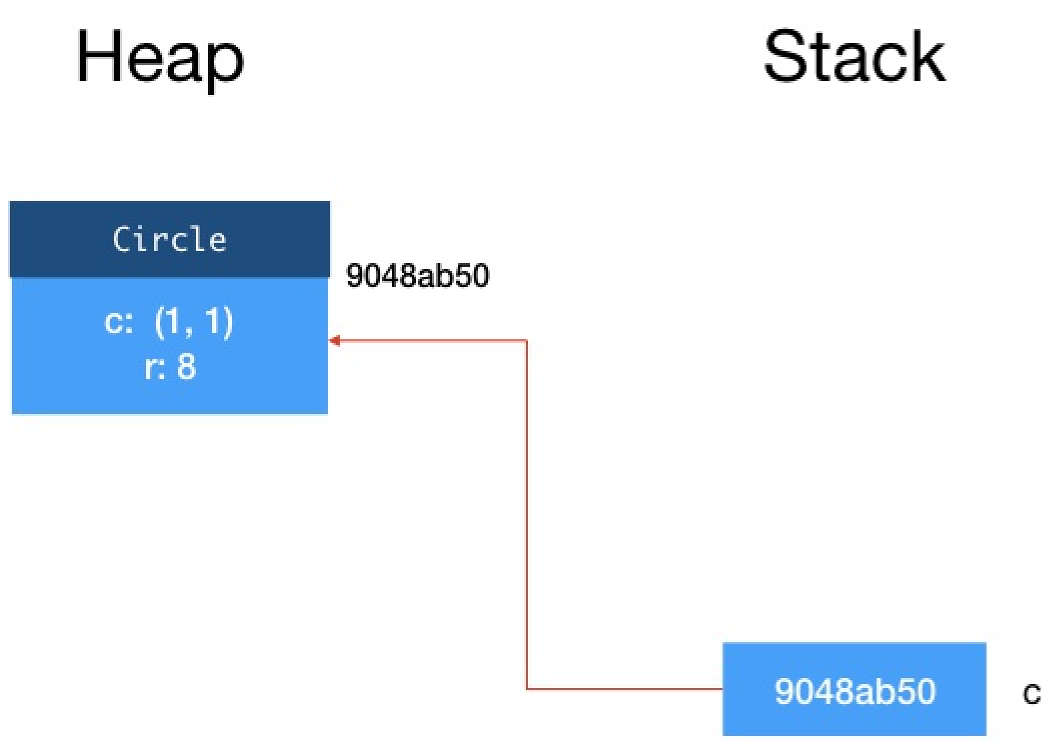
Local classes:

- Refactored during compilation into a normal class
- A `final` field is added for each captured variable
- A reference to the outer class (so that the local class instance can access fields of the associated outer class instance) is added.
- Thus, the new normal class can function as if it is a local class.

Notes:

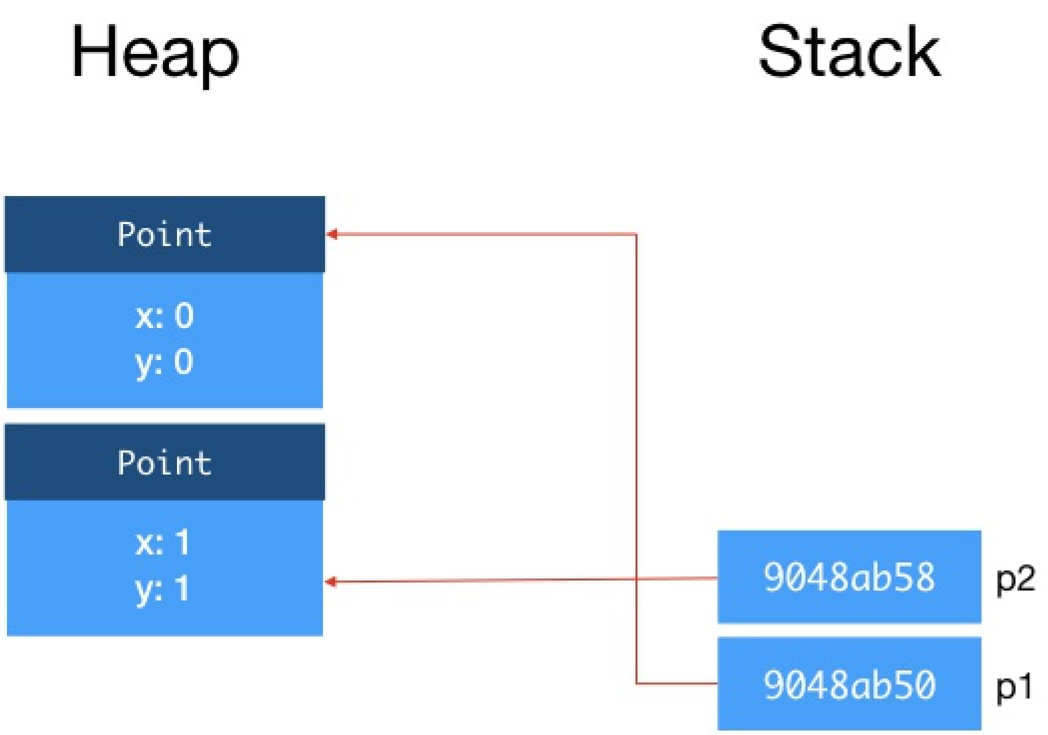
- Class definitions never reside on the stack or heap
- For static methods, there is no "this" reference variable on the stack because a static method is not associated to a specific instance of the class
- JVM keeps a *method* area for storing the code for the methods; - *metaspace* for storing meta information about classes; - *heap* for storing dynamically allocated objects; - *stack* for local variables and call frames.
- `null` means that a reference is not pointing to any object.

Stack and Heap Diagram v2



Method call

Before

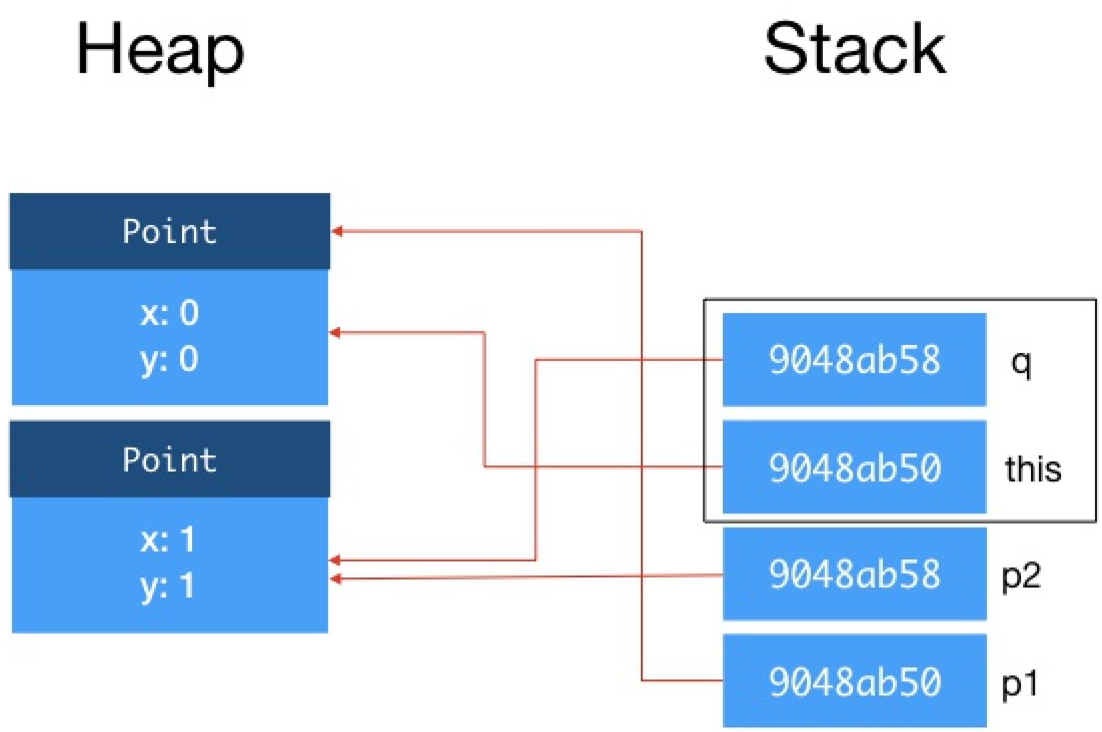


JVM creates a *stack frame* for this instance method call. This stack frame contains

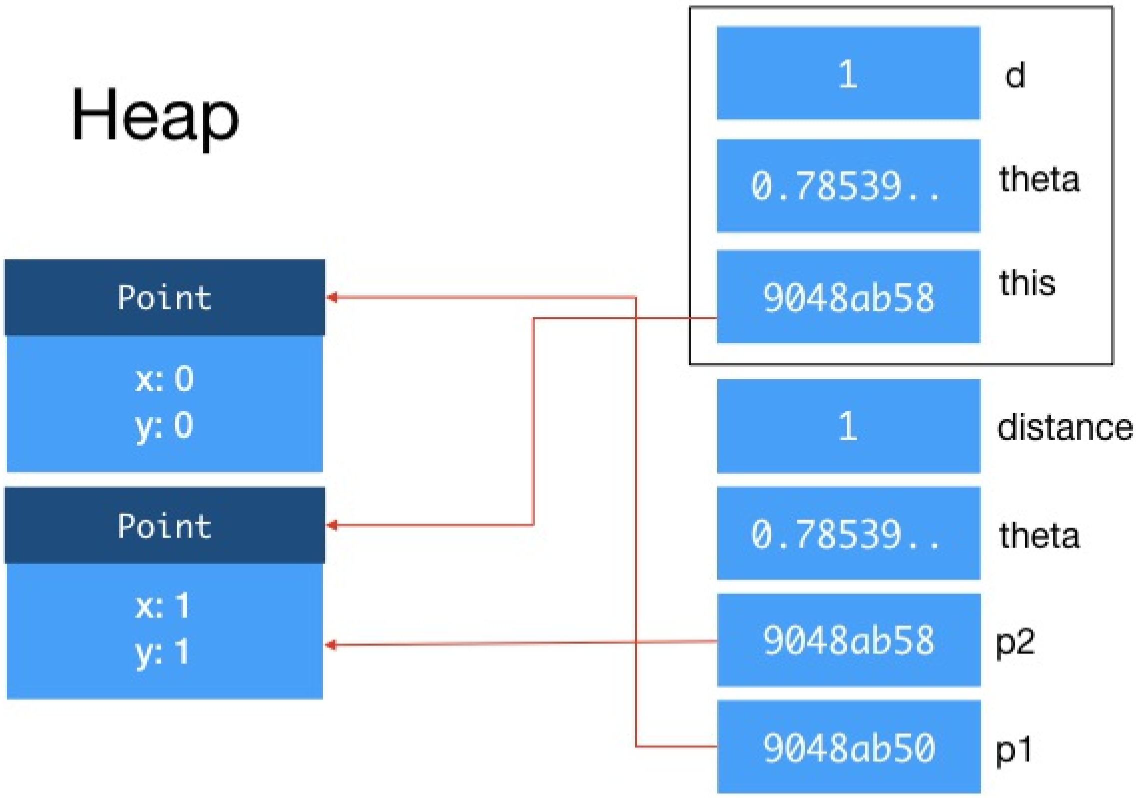
1. `this` reference.
2. The method arguments. `q`
3. Local variables within the method. *(Not shown)*

*When a class method is called, the stack frame does not contain the `this` reference. *

After



With Primitives



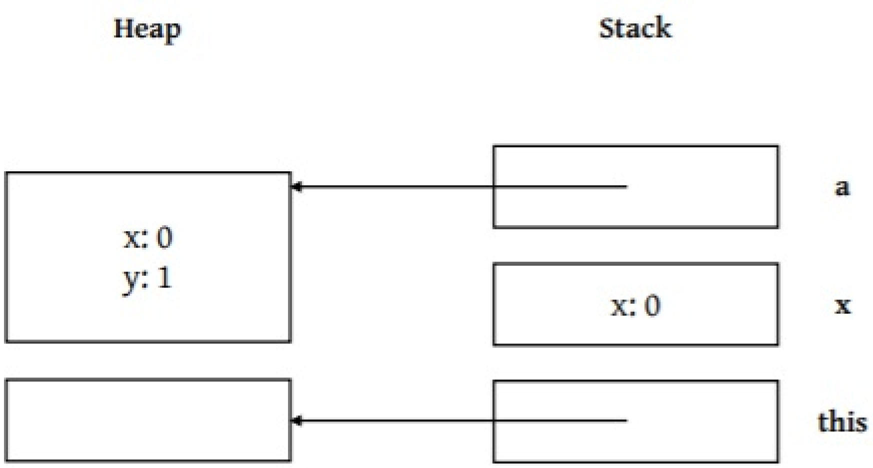
Note that `d` and `theta` do not point to an object but instead are passed by value.

Variable capture

Consider the program below :

`Java class B { void f() { int x = 0; class A { // This is a local class int y = 0; A() {y = x + 1;} } A a = new A(); } }`

Suppose that a variable b is an instance of class B, and a program calls b.f().
Sketch the content of the stack and heap immediately after the
Line A a = new A() is executed. Label the values and variables / fields clearly.
You can assume b is already on the heap and you can ignore all other content of
the stack and the heap before b.f() is called.



Variable capture: Local class makes a copy of local
variables used from the enclosing method to within itself
This stack frame contains (due to method call):

1. `this` reference.
2. No method arguments.
3. Local variables within the method. `x`

This stack frame contains (due to variable declaration):

1. `a`, the variable initialised with a `new A()` object.

This heap contains (due to variable capture):

1. An instance of class `A`.
    i. Captured variable `x` now part of its instance attributes.
    ii. Declared variable `y` now part of its instance attributes.

Types

There are 2 types in Java: Primitive and Composite (usually in the form of an ADT)
There at a few kinds of variables:

- Instance variables (non static fields)
- Class variables (static fields)
- Local variables (see Variable Capture)
- Parameters (these are *not* fields)

Typing and Variance

- To denote a subtype relation e.g `S` is a subtype of `T`, we say `S <: T`.
- For primitive: `byte <: short <: int <: long <: float <: double; and char <: int`.

Suppose `A(T)` is the complex type constructed from `T`. Then we say that

- A is covariant if `T<:S` implies `A(T)<:A(S)`,
- A is contravariant if `T<:S` implies `A(S)<:A(T)`,
- A is bivariant if it is both covariant and contravariant,
- and A is invariant if it is neither covariant nor contravariant.

Reference Conversion

Primitive Conversion

Liskov substitution principle

If `S` is a subtype of `T`, then objects of type `T` may be replaced with objects of type `S` (i.e. an object of type `T` may be substituted with any object of a subtype `S`) without altering any of the desirable properties of `T`.

How to answer question:

- Show the property f(`S`) that is not present in `T`, i.e f(`T`) does not hold true
- Say that if an instance of `S` is replaced by an instance of `T`, this property will not hold
- Thus, LSP is violated

Method matching

There are 3 steps Java uses to find the method to fit, and after that prioritises more accurate types.

- The first step allows for implicit widening conversions.
- The second step allows for auto-boxing and unboxing (in addition to those in step 1).
- The third step allows for variable arity methods (in addition to those in steps 1 and 2).

Within a step, if any applicable methods were found, the proceeding steps will be skipped. If multiple applicable methods were found, the most specific method will be selected. If there are more than 1 most specific methods, the method invocation is *ambiguous* and you get a compile-time error.

Null

> There is also a special null type, the type of the expression null, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type. The null reference is the only possible value of an expression of null type. **The null reference can always be cast to any reference type.** In practice, the programmer can ignore the null type and just pretend that null is merely a special literal that can be of any reference type.

- The null reference can always be cast to any reference type.
- A variable cannot be declared to be type null.
- Null is technically a reference type.

Generics & Collections

Generics & Type Erasure

Suppose a class or interface `B` is a subtype of `A`, then `B<T>` is also a subtype of `A<T>`, i.e., they are covariant.

Generics, however, are *invariant*, with respect to the type parameter. That is, if a class or interface `B` is a subtype of `A`, then neither is `C<B>` a subtype of `C<A>`, nor is `C<A>` a subtype of `C<B>`. A parameterized type must be used with exactly with same type argument. For instance: `Queue<Integer>` is *not* a subtype of `Queue<Object>`.

Type Erasure:

- `Queue<Circle>` will be replaced by `Queue` and `T` will be replaced by `Object`.
- `Queue<? extends Shape>` will be replaced with `Queue` and `T` will be replaced with `Shape`
- The compiler also inserts type casting and additional methods to preserve the semantics of the generic type
- We cannot have both void foo(Queue c) {} and void foo(Queue c) {} as they both get converted to void foo(Queue c) {} in compilation
- Both Queue and Queue will share static methods
- Queue q = new Queue() is thus legal as the code will eventually become `...new Queue()` after type erasure.
- We can explicitly cast supertypes and assign them to subtypes, but this might cause an `Exception` later on e.g `ClassCastException`.
  Conclusion: In runtime, the **type information is not available**.

Wildcards

Typing:

- `List<A>` is a possible subtype of `List<? extends A>`

Initialisation/Assignment with wildcard:

- Initialisation: A declared type of `LinkedList<? extends T>` or `LinkedList<? super T>` will lead to an object type of `LinkedList<T>` being inferred.
- Assigning `A<Object>` to `A<? extends Object>`: The parameterised type is bounded by `? extends Object`.
- Assigning `A<Integer>` to `A<? super Integer>`: Java infers the type to be `Integer`.

Getting and setting from references with wildcard generics:

- We can get `A` typed items from `List<? extends A>` as any item will be extending `A`, so implicit widening reference conversion will help us assign the object, that extends `A`, to a variable of type `A`.
- We can add `A` typed objects into a `List<? super A>` as any list it refers to will have a parameterised generic supertype of `A` (or `A` itself). So, implicit widening reference conversion will help us add an `A` typed object into the list.

Exceptions

Rules

Any code that might throw `Exception`s must either *Catch* or *Specify*.

- Catch: A `try` statement that catches the `Exception`. It must provide a handler for this `Exception`
- Specify: A method can specify that it can `throw` an `Exception`.
- In essence, *we need to either catch all checked exceptions or let it propagate to the calling method*. Code that fails to honor the *Catch* or *Specify* Requirement will not compile.

Good Practices

- Preferrably, more specific `Exception`s should be thrown, to prevent loss of information and to prevent unintended catching of `Exception`s.
- Make sure to catch exceptions to clean up/deallocate resources.
- Don't expose implementation through the thrown `Exception`. You can always make a wrapper.
- Don't do "Pokemon" catching.

Notes

- The `finally` block *always* executes when the `try` block exits.
- Unchecked `Exception`s are not required to be specified in the method.
- When overriding a method, the new method must throw the same, or more specififc (subclass) `Exception` as the overriden method.
- All unchecked exceptions are implicitly declared to be thrown by any method e.g `Error` and `RuntimeException`.
- The `catch` block is not just limited to `Exception`s: any class that inherits from `Throwable` can be caught there.
- You can use the `|` operator to group a few specific `Exception`s together e.g `catch (IOException|SQLException ex) {...}`

hashCode, Nested Class, enum, variable capture

hashCode

Remember to override `hashCode()` and `equals()` if you are goinf ot use a `HashMap` or a `HashSet`.

Type safety (relates to Generics as well)

- enum allows a type to be defined and used for a set of predefined constants. Using a constant other than those predefined would lead to a compilation error. In contrast, using int is not type safe since int values other than those predefined can be accidentally assigned / passed as arguments.
- Generics allow classes / methods that use any reference type to be defined without resorting to using the Object type. It enforce type safety by binding the generic type to a specific given type argument at compile time. Attempt to pass in an incompatible type would led to compilation error

Variable Capture & Local Class

- A local class has access to the members of its enclosing class.

- A local class has access to local variables. How ever, a local class can only access local variables that are declared final or *effectively* final. When a local class accesses a local variable or parameter of the enclosing block, it captures that variable or parameter. Note that instance variables are not captured, only local variables.
- Local classes are similar to inner classes because they cannot define or declare any static members.
- Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

Default Initialisation

Data Type	Default Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'`
`String (or any object)`	`null`
`boolean`	`false`