CS2030S cheatsheet (1)

jovyntls

TYPES

- **S is a subtype of T,** `S <: T` if a piece of code written for variables of type `S` can also be safely used on variables of type `T`.
  - *widening* conversion ⇒ a type `S` can be put into a variable of type `T` if `S <: T`.
  - *narrowing* conversion ⇒ requires typecasting
- **reflexive** - `T <: T`
- **transitive** - if `S <: T` and `T <: U`, then `S <: U`
- `S instanceof T` returns true if `S <: T`

primitive types

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

Liskov substitution principle

- if `S <: T`, then
  - any property of `T` should also be a property of `S`. (includes fields, methods)
  - an object of type `T` can be replaced by an object of type `S` without changing some desirable property of the program.
- **VIOLATION:** subclass changes the behaviour of the superclass - <specified> property no longer holds.
  - places in the program where the superclass is used cannot be replaced by the subclass

RUN-TIME vs COMPILE-TIME TYPES

```
Circle c = new ColouredCircle(p, 0, red);
// ColouredCircle <: Circle
```

- compile-time type: `Circle`
- run-time type: `ColouredCircle`

OOP PRINCIPLES

encapsulation

- composite data types
- abstraction barrier - hide information & implementation
- `private` attributes, `public` methods

inheritance

- "is-a" relationship → `extends` (subtyping)
- vs "has-a" relationship → use composition

tell, don't ask

- don't make assumptions the implementation
- a class should be agnostic of another class

polymorphism

- **dynamic binding** → method invoked is determined at runtime

method overriding

- same method signature (method name + type of arguments)
- dynamic polymorphism

method overloading

- same method name, diff parameter types/number of parameters
- static polymorphism

ABSTRACT CLASSES

```
abstract class Shape { ... }
```

- cannot be instantiated
- a concrete class cannot have abstract methods
  - as long as one method is abstract, the whole class is abstract
- an abstract class can have concrete and/or abstract methods

INTERFACE

```
interface getAreable {
  // methods are public and abstract by default
  double getArea();
}
```

- abstract class
  - concrete classes implementing the interface have to implement the body of the methods
- if class `C` implements interface `I`, then `C <: I`.
- a class can extend multiple interfaces

  ```
  class C implements A, B { ... }
  ```

- an interface can extend multiple interfaces

  ```
  interface I extends A, B { ... }
  ```

  - an interface cannot implement other interfaces (abstract!!)

💡 `this` - reference variable that refers to the instance

CS2030S cheatsheet (2)

jovyntls

WRAPPER CLASS

```
Integer i = new Integer(2);
int j = i.intValue();
```

(un)boxing

```
int i = 1;      // i is an int
Integer j = i;  // j is an Integer
int k = j;      // k is an int
```

JAVA

access modifiers

`private` → only within the class

`default` → only within the package

`protected` → only within the package or outside the package through the child class

`public` → everywhere

`final` keyword

- `final` class → cannot be inherited from
- `final` method → cannot be overridden

CASTING

```
// Circle <: Shape <: GetAreable
GetAreable findLargest(GetAreable[] array) { ... }
GetAreable ga = findLargest(circles);       // ok
Circle c1 = findLargest(circles);           // error
Circle c2 = (Circle) findLargest(circles);  // ok
```

- only cast when you can prove that it is safe

variance

Let $C(T)$ be a complex type based on type T. The complex type C is:

- **covariant** if $S <: T$ implies $C(S) <: C(T)$
- **contravariant** if $S <: T$ implies $C(T) <: C(S)$
- **invariant** if C is neither covariant nor contravariant

(Java array is covariant)

EXCEPTIONS

```
try {
  new Circle(new Point(1, 1), 0);
  // everything afterwards is skipped
  System.out.println("This will never reach");
} catch (IllegalException e) {
  // runs if there is an exception
} finally {
  // always runs
}
```

- exception will be passed up the call stack until it is caught
- after exception is caught: everything else proceeds normally

`throw` exceptions

```
public Circle(Point c, double r) throws IllegalArgumentException {
    if (r < 0) {
      throw new IllegalArgumentException("radius cannot be negative.");
    }
    // anything from here will not run if r<0
}
```

- `throw` causes method to immediately return

CS2030S cheatsheet (3)

jovyntls

GENERICS

- allow classes/methods (that use reference types) to be defined without resorting to using the Object type.
  - ensures **type safety** → binds a generic type to a specific type at compile time
  - ✓ errors will be at compile time instead of runtime
- generics are **invariant** in Java

generic class

```
class Pair<S extends Comparable<S>, T> implements Comparable<Pair<S, T>> {...}
class DictEntry<T> extends Pair<String,T> {...}
```

generic method

```
public static <T> boolean contains(T[] arr, T obj) {...}
// to call a generic method:
A.<String>contains(strArray, "hello");
```

- type parameter `<T>` is declared *before* the return type

note

```
B implements Comparable<B> { ... }
A extends B { ... }
A <: B <: Comparable<B>
Comparable<A> INVARIANT Comparable<B>
Comparable<A> <: Comparable<? extends B>
```

TYPE ERASURE

type erasure

- at compile time, type parameters are replaced by `Object` or the bounds (e.g. `T extends Shape` is replaced by `Shape`)

suppress warnings

- `@SuppressWarnings` can only apply to declaration

```
@SuppressWarnings("unchecked")
T[] a = (T[]) new Object[size];
this.array = a;
```

WILDCARDS

upper-bounded: `? extends`

- *covariant* - if `S <: T`, then `A<? extends S>` <: `A<? extends T>`

lower-bounded: `? super`

- *contravariant* - if `S <: T`, then `A<? super T>` <: `A<? super S>`

unbounded: `?`

- `Array<?>` is the supertype of all generic `Array<T>`

PECS PRINCIPLE

> 💡 Producer `extends`; Consumer `super`

- `PE` → if you need to produce T values, declare `List<? extends T>`
- `CS` → if you need to consume T values, declare `List<? super T>`

RAW TYPES

- a generic type used without type arguments
- only acceptable as an operand of `instanceof`

TYPE INFERENCE

- ensures **type safety** → compiler can ensure that `List<myObj>` holds objects of type `myObj` at compile time instead of runtime
- `<? super Integer>` ⇒ inferred as `Object`
- `<? extends Integer>` ⇒ inferred as `Integer`

diamond operator: `<>`

```
Pair<String,Integer> p = new Pair<>();
```

- only for instantiating a generic type - not as a type
- generic methods: type inference is automatic
  - `A.contains()` not `A.<>contains()`

constraints for type inference

- target typing → the type of the expression (e.g. `Shape`)
- type parameter bounds → `<T extends GetAreable>`
- parameter bounds → `Array<Circle>` <: `Array<? extends T>`, so `T :> Circle`

```
public static <T extends GetAreable> T findLargest(Array<? extends T> array)
Shape o = A.findLargest(new Array<Circle>(0));
```