abstract classes and methods

- cannot be instantiated unless inherited from
- a class with at least one abstract method must be declared abstract. On the other hand, an abstract class may have no abstract method.

```
abstract class Task {
 private String description;
 private boolean isCompleted = false;
 public Task(String description) {
  this.description = description;
 }
 public void complete() { }
 public boolean isCompleted() { }
 public abstract boolean isDueToday();
 public abstract int getRewardPoints();
 public abstract void remind();
 public String toPrettyString() { }
 @Override
 public String toString() { }
}

public class AT extends Task {
 private String assign;
 public AT(Str tt, String assign){
        super(tt); //use super constructor
        String assign = assign;
 }
 @Override //override methods which
behaviour needs to be changed
 public void remind() { overridden; }
}
```

generics

```
public class ArraySt<T> implements St<T> {
        private int maxStackSize;
        private T[] stack;
        private int currentIndex = 0;

        public ArrayStack(int maxStackSize) {
                this.maxStackSize = maxStackSize;
                // comment on why suppress warnings is ok
                @SuppressWarnings("unchecked")
                T[] temp = (T[]) new Object[maxStackSize];
                this.stack = temp;
        }
}
```

<? extends T> - upper bounded wildcard
<? super T> - lower bounded wildcard

narrowing type conversion requires an explicit type cast to be made

exceptions

try-catch-finally blocks, if we actually want to do something with the error

```
private TaskList(Scanner sc) {
 try {
  loadTasks(sc);
 } catch (WrongTaskTypeException e) {
  System.out.println(e.getMessage());
 //buck passed ^
 } finally {
  sc.close();
 }
}

private void loadTasks(Scanner sc) throws
WrongTaskTypeException { loadTasksB; }

private void loadTasksB(Scanner sc) throws
WrongTaskTypeException { . . . throw new
WTTException("message"); }
```

```
//creating exceptions

class ExceptionA extends SuperClassException {
       public ExceptionA(String msg) {
        super(msg); //most likely what will happen
       }
}
```

unchecked exceptions are not explicitly caught or thrown. cause run-time errors.

checked exception is an exception that a programmer has no control over.

```
class A {
  public static <T> boolean
c(T[] a, T obj) {
    for (T curr : a) {
      if (curr.equals(obj)) {
        return true;
      }
    }
    return false;
  }
} //GENERIC METHOD
```

check for diamond brackets in code in order to confirm whether generics are used or not - if it seems like generics but no <> then it may be wise to use Object

LSP

- an overriding method has to throw an error that is the same is the method being overridden or its subtype
- a subclass cannot violate the behaviour of the parent class

`final`:
In a class declaration to prevent inheritance.
In a method declaration to prevent overriding.
In a field declaration to prevent re-assignment - important for constants

interfaces

abstraction to model what an entity can do

- all methods in an interface are <u>public abstract by default (unless otherwise declared)</u>
- for a class to implement an interface and be concrete, it has to <u>override all abstract methods from the interface and provide an implementation to each</u>. otherwise, the class becomes abstract*

```
interface Stack<T> {
        T pop();
        void push(T item);
        int getStackSize();
} //generic interface
```

OOP

typically model the nouns as classes and objects, the properties or relationships among the classes as fields, and the verbs or actions of the corresponding objects as methods.

tell don't ask

composition models the HAS-A relationship between two entities (e.g. a Circle has a point)

inheritance models the IS-A relationship on the other hand (e.g. a Circle is a shape)

model out these relationships properly before even typing any code, DO NOT RUSH!!! REFACTORING SUCKS.

spare notes/motivation: u got dis stranger!

@qilstiano on Github