Exceptions

- Illegal arg. Exception.
  - ↳ method is inappropriately passed.
  - ↳ unchecked.
    Failure of programmer.
  - ↳ throw using constructor.
    - Specify that method can throw the exception.
    - Keyword "throws" on method. "throw" inner.
    - Specify detailed msg.
      eg. "Radius not positive".
    - Can extend it ⇒ make it more accurate.

Generics

- prevent redundant code
- rather than having a specific type
  - ↳ take advantage of class hierarchy?
    - Object.
    - no need to worry about type, get away with creating one class
    - Can change things in one place.
    - Isn't perfect
    - Eg.

```
Pair p = new Pair(1, "Hello World")

Integer i = p.getFirst();
```

runtime type of p

⇓ Incompatible types. Obj ⇒ Int.

⇓ pair class returns Object

∴ narrow conversion.

∴ Cast to (Integer).

// works.

⇓ What if cast wrong? Compiler ignores warning.

⇓ will compile without issue...

⇓ runtime error: class cast exception.

issue.

Generic Types

- Create a class w/ type parameters (like variable).
  Can pass to them what type you want.
- eg.

```
class Pair<T, S> {
    private T first;
    private S second;
    .
    .
    . Same everywhere type is involved
    replace obj w/ 2 diff type params.
}
```

To instantiate:

```
Pair<String, Integer> p = new Pair<String, Integer>("Hello", 3);

String s = p.getFirst();
```

also can ignore this and let java infer the types.

↳ no need type casting.

↳ mistakes are caught in Compile time.

class DictEntry<T> extends Pair<String, T>{}

key, value. — eg. Circle — fixed. — chained together.

belong to the method itself. (not the class).

generic method — public static <T> boolean contains(T[] array, T obj);

- can use .equals method.
  - subtype - object.
- String[] strArray = new String[]{...};
- Specify A.<String>contains(strArray, "Hello");
  => true / false.

incompatible if
A.<Integer>contains(strArray, "Hello");
- catch at CT.

Bounded Type parameters.

```
class B{
    public static <T> T findLargest(T[] arr){
        double maxArea = 0;
        T maxObj = null;
        for(T curr: array){
            :
        }
    }
```

before return type.

Can't find a getArea() method on type T.

\ how to tell the compiler that the type param. need to be a subtype of smth else.

public static <T extends GetArea> T findLargest(T[] array){
∴ T <: GetArea
:

bounded type params

Constraint of type allows us to enforce subtype relationship.
∴ take advantage of the relationship.

Comparable is a generic interface

- Impose a total ordering on each element of a class.
- Can compare which one comes first.
- Interface with us to specify one method -

int compareTo(T o)

Is -ve int if less than
Is 0 if same
Is +ve if greater.

- class Integer implements Comparable<Integer>

```
@Override
public int compareTo(Pair<T, S> p)
    this.first.compareTo(cp.first);
}
```

↳ no compareTo method
↳ need to specify it is bounded by
a subtype of the comparable interface

∴ class Pair<T extends Comparable<T>, S> implements Comparable<Pair<T,S>>{

;

Sort → natural ordering → compareTo method.

/
Sort it by lst/snd
change here.

Problems w/ Generics

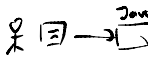
— Implementation in Java.

Generic Pair class

↳ using Object
(how people used to
do it without generics)

↳ in creating generics, have to
be backwards compatible.

Type Erasure



- type checking
- type erasure process.

} process done by the Compiler

need to be aware of.

Implement generics?

- Code specialisation

generate a new class for
every new type arguments
by the compiler.

↳ many end up creating
hundred of classes.

- Code sharing. (by Java) *

↳ type checking first
erase the types and type params.

Compilation (In Javac)
↳ type checking

eg: `pair<String, Integer> = new Pair<String, Integer>('...');`

♡ Check if types line up nicely
⇒ sure about their types.
⇒ remove all the type params.

♡ `Pair p = new Pair(...)`

get rid of all generic types.
replace them w/ Object.
⇒ look at the original pair class
⇒ but type is checked.
⇒ hence it is safe.

What if type param is bounded?

eg. `S extends Comparable<S>`

unbounded types are replaced w/ Object.
bounded types are replaced w/ bound.

eg `Integer i = new Pair<S,T>().`
`Integer i = (Integer) new Pa`

(

compiler need cast
⇒ type erasure
cannot cast for you.

Strange behaviors ⇒ reification

Important to note: only 1 class but
when I use type info
⇒ no longer can infer these types

Pair<S, T>[] pairArr.

Cannot mix arrays w/ generics // error.

Object[] objArr.

pairArr = new Pair[2];

Set ObjArr = pairArr.

Since Pair <: Object

∴ Covariance.

∴ pair array is subtype of Object

objArr[0] = new Pair(3.14, true);

String str = pairArray[0].getFirst();

Integer.

ArrayList<E>

generic Array Abstraction.

To handle both type:

```
class Array<T> {
    private T[] array;
    public Array(int length){
        @SuppressWarnings("unchecked")
        T[] temp = (T[]) new Object[length];
        this.array = new T[length];
                   = (T[]) new Object[length]
    }
    public set(int index, T t){
        this.array[index] = t;
    }

    public T get(int index){
        return this.array[index];
    }
    public T[] getArray(){
        return array;
    }
}
```

only can do it on declaration

Can also do it on method or class but not recommended

leave a comment. // only way to get things into array is through set method ∴ safe to cast.

not possible.

can't instantiate.

warning! unchecked cast. (erased)

enforces type T.

Suppress warning.

Raw Types.

possible to use Array class w/o <?>.

but no info about parameterized type.

⇒ go in and try put in a number.

⇒ compiler won't catch for us.

⇒ dangerous.

⇒ use only for instanceOf eg. Comparable

|

only happens at runtime

⇒ no type param info.