Java refresher Cheat Sheet

by nixik09 via cheatography.com/212204/cs/46107/

Data types in Java

Type	Size (bits)	Min	Max	Ex
byte	8	-2^7	-2^7 -1	byte b = 100;
short	16			
int	32			
long	64			
float	32			
double	64			
char	16			
boolean	1			

Checked and unchecked exceptions and errors

Why "(un)checked"? Compiler can't anticipate logical errors that arise only at runtime, can't *check* for those types of problems -->"unchecked exceptions". Typically unchecked comes from logical errors/faulty logic that can occur anywhere.

Checked Exceptions

- Exceptional conditions that an app should anticipate and recover from.

- E.g. `FileNo tFo und Exc eption` occurs when a method is trying to read a file that does not exist

- Checked at compile time - should be stated in method signature if throwing an exception. If exception could potentially be thrown in code, must handle it too.

Does not inherit from `Runtim eEx ception` or `Error`? then its a **checked exception**

Unchecked Exceptions

- Exceptional conditions that app cannot anticipate/recover from

- E.g. is `NullPo int erE xce ption` - when method is expecting non-null value but receives null.

- Not checked at compile time. Not required to be in method sig, not required to be handle it in code.

Is inherited from `Runtim eEx ception`? then its an **unchecked exception**

Errors

Checked and unchecked exceptions and errors (cont)

E.g. `OutOfM emo ryError` - when app is trying to use more memory than avail on system

- Not checked at compile time and not usually thrown from app code.

Switch statements

```
String direction = getDirection();
switch (direc tion) {
        case " lef t":
                goL eft();
                break;
        case " rig ht":
                goR ight();
                break;
         def ault:
                return " unk now n";
}
// Java 14+
return switch (shirtNum) {
          case 1 -> " goa lie ";
          case 2 -> "left back";
          case 3, 4 -> " centre back";
          case 6, 7, 8 -> " mid fie lde r";
           default -> throw new Illega lAr gum -
ent Exc ept ion ("In valid shirt number: " +
shirtNum);
}
```

- Default case optional, but largely good practice to include one

Streams

- represents *sequence of elements* and operating on those elements. Not data structures but take input from collections, arrays, or I/O channels.

- key benefits: declarative, readable code; parallel operations; builtin operations; less boilerplate

Streams (cont)

Creating from a collection	List<String> list = Arrays.asList("apple", "banana", "cherry"); Stream<String> streamFromList = list.stream();
Creating a stream from an array	String[] array = {"apple", "banana", "cherry"}; Stream<String> streamFromArray = Arrays.stream(array);
Creating stream using Stream.of	Stream<String> streamOfElements = Stream.of("apple", "banana", "cherry");
Creating empty stream	Stream<String> emptyStream = Stream.empty();
Creating infinite streams	Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);

Primitive streams

`mapToInt()` vs `map()`

- `map(St rin g:: length)` returns a `Stream <In teg er>` (stream of `Integer` objects)
- `mapToI nt( Str ing ::l ength)` returns an `IntStream` (stream of primitive `int`s)

Specialised primitive streams

- `IntStream`, `LongStream`, `Double Stream`

These streams have additional operations not available on regular streams:

- sum: `intStr eam.sum()`
- average: `intStr eam.av erage()`
- statistics: `intStr eam.su mma ryS tat ist ics()`

Using primitive streams avoids boxing/unboxing overhead when dealing with numeric operations

Another stream ex.

```
List<Person> people = Arrays.asList(
        new Person ("Jo hn", 25),
        new Person ("Sa rah ", 32),
        new Person ("Mi ke", 17),
        new Person ("Em ily ", 25)
);
```

Another stream ex. (cont)

```
> // Find names of adults, sorted alphabetically
List<String> adultNames = person.stream()
    .filter(person -> person.getAge() >= 18)
    .map(Person::getName)
    .sorted()
    .collect(Collectors.toList());
System.out.println(adultNames); // [Emily, John, Sarah]
double averageAge = people.stream()
    .mapToInt(Person::getAge)
    .average()
    .orElse(0.0);
System.out.println("Average age: " + averageAge);
```

Method references

Syntax	Equivalent Lambda	Meaning
object::instanceMethod	x -> object.instanceMethod(x)	Use object as the target for each call
Class::staticMethod	x -> Class.staticMethod(x)	Call a static method
Class::instanceMethod	(obj, arg) -> obj.instanceMethod(arg)	Useful in sorting or grouping

Concurrency: creating threads

```
// 1. by inheriting from Thread class
public class Exampl eThread extends Thread {
        @Ov erride // note override! invoked when thread starts
        public void run() { // we do NOT call RUN!!
                // contains all the code to execute when starting thread
                Sys tem.ou t.p rin tln (Th rea - d.c urr ent Thr ead ().g et Nam e());
        }
}
// start new thread..
public class Thread Exa mples {
        public static void main(S tring[] args ) {
                Exa mpl eThread thread = new Exampl eTh read();
```

Concurrency: creating threads (cont)

```
>       thread.start(); // only START to start new thread
    }
}
// 2. implementing Runnable interface
public class ExampleRunnable implements Runnable {
    @Override // note OVERRIDE!!
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
public class ThreadExamples {
    public static void main(String[] args) {
        ExampleRunnable runnable = new ExampleRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
// 3. Anon declarations
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println(Thread.currentThread().getName());
        });
        thread.start();
    }
}
```

Both 1) and 2) work exactly the same with no diff in performance. BUT, Runnable interfaces leaves option of extending class with some other class since you can inherit only one class in Java. Also, easier to create a thread pool using runnables.

Throwing and handling exceptions

```
/**
Throwing a checked exception
*/
public class Insuff ici ent Bal anc eEx ception
extends Exception {}
public class BankAc count {
        public void withdr aw( double amount)
throws Insuff ici ent Bal anc eEx ception {
                if (balance < amount) {
                        throw new Insuff ici ent -
Bal anc eEx cep tion();
                }
        }
/**
Throwing an unchecked exception
*/
public class BankAc count {
        public void withdr aw( double amount) {
                if (amount < 0) {
                        throw new Illega lAr gum -
ent Exc ept ion ("Cannot withdraw a negative
amount ");
                }
        }
}
/**
try, catch and finally
*/
try {
        ban kAc cou nt.w it hdr aw( amo unt);
} catch (Insuf fic ien tBa lan ceE xce ption) {
        Sys tem.ou t.p rin tln ("Wi thd rawal
failed: insuff icient balanc e");
} catch (Runti meE xce ption e) {
        Sys tem.ou t.p rin tln ("Wi thd rawal
failed: " + e.getM ess age());
} finally {
        Sys tem.ou t.p rin tln ("Cu rrent balance:
" + bankAc cou nt.g et Bal anc e());
}
```

Java Maps

dictionary DS - HashMap, TreeMap

Map<String, Integer> fruitPrices = new HashMap<>();

Add entries	fruitPrices.put("apple", 100);
Get value for a key	fruitPrices.get("apple");
Check if map contains specific key	fruitPrices.containsKey("apple"); // => true
Remove entries	fruitPrices.remove("plum");
Get size	fruitPrices.size();
Get all keys in map	fruitPrices.keySet(); // returns keys in a set
Get all values in map	fruitPrices.values(); // returns values in a collection

Stream intermediate operations

```
List<String> fruits = Arrays.asList("apple",
"banana", "cherry", "date");
// filter - keeps elements that match a predicate
Stream <St rin g> longFruits = fruits.st ream()
        .fi lte r(fruit -> fruit.l en gth() > 5);
// map - transforms each element
Stream <St rin g> fruitL engths = fruits.st ream()
        .ma p(S tri ng: :le ngth);
// sorted - sorts elements
Stream <St rin g> sorted Fruits = fruits.st ream()
        .so rted();
// distinct - removes duplicates
Stream <St rin g> unique Fruits = fruits.st ream()
        .di sti nct();
// limit - reduces stream size
Stream <St rin g> limite dFruits = fruits.st ream()
        .li mit(2);
// skip - skips elements
Stream <St rin g> skippe dFruits = fruits.st ream()
        .sk ip(1);
```

- These return a new stream and are lazy for performance reasons
- Allows JVM to optimise entire operation chain at once
- For ex., if you filter 1000 elements and then limit to 5, Java can stop processing after finding 5 elements that match the filter (rather than filtering all 1000 first).

Stream terminal operations

```
List<String> fruits = Arrays.asList("apple",
"banana", "cherry");
// forEach - performs actions on each element
fruits.st rea m().fo rEa ch( Sys tem.ou t:: pri -
ntln);
// collect - puts each element into a collection
List<S tri ng> fruitList = fruits.st ream()
        .fi lte r(fruit -> fruit.l en gth() > 5)
        .co lle ct( Col lec tor s.t oLi st());
// reduce - reduces stream to single value
Option al< Str ing> combined = fruits.st ream()
        .re duc e((a, b) -> a + ", " + b);
// count - returns number of elements
long count = fruits.st rea m().co unt();
// anyMat ch/ all Mat ch/ non eMatch - check
predicates
boolean anyLong = fruits.st ream()
        .an yMa tch (fruit -> fruit.l en gth() >
5);
boolean allLong = fruits.st ream()
        .al lMa tch (fruit -> fruit.l en gth() >
3);
boolean noneLong = fruits.st ream()
        .no neM atc h(fruit -> fruit.l en gth() >
10);
// findFi rst /fi ndAny - find elements
Option al< Str ing> first = fruits.st rea m().fi -
ndF irst();
```

- These are terminal - they **don't return a stream** but **return a concrete result or side-effect** (e.g. collection, primitive or object).
- They trigger the actual processing of stream elements - note above collect() returning a collection, count() returning a long, reduce() returning an Optional/specific value or forEach() returning void (producing side effects).
- `average()` - operation on `IntStream` - returns OptionalDouble

Method references code samples

```
- String::length and x::equals - not static
methods; equals() and length() are both instance
methods.
- The shorthand forms are method refere nces,
equivalent to x -> operat ion.eq uals(x) or x-> -
x.l ength.
//object ::i nst anc eMethod
```

Method references code samples (cont)

```
> String prefix = " Hel lo";
List<S tri ng> words = List.o f("H ell o", " Hi", " Hey ");
boolean allMatch = words.s tr eam()
    .allMa tch (pr efi x:: equ als); // same: x -> prefix.eq uals(x)
//Class: :st ati cMethod
// Use for static utility methods, like from Math or Integer
List<S tri ng> numbers = List.o f("1 ", " 2", " 3");
List<I nte ger> ints = number s.s tream()
   .map(I nte ger ::p ars eInt) // same as: x -> Intege r.p ars eInt(x)
   .toList();
List<D oub le> values = List.o f(9.0, 16.0, 25.0);
List<D oub le> roots = values.st ream()
   .map(M ath ::sqrt) // same as: x -> Math.s qrt(x)
   .toList();
//Class: :in sta nce Method
List<S tri ng> items = List.o f("H ell o", " Hi", " Hey ");
List<S tri ng> lower = items.s tr eam()
   .map(S tri ng: :to Upp erCase) // same as: x -> s -> s.toUp per Case()
   .toList();
List<S tri ng> nonEmpty = items.s tr eam()
   .filter(s -> !s.isE mpt y()); // classic
// or
List<S tri ng> nonEmpty2 = items.s tr eam()
  .filter(Predicate.not(String::isEmpty)) // java 11+
  .toList();
//Constr uctor references - ClassN ame ::new
List<S tri ng> list = Stream.of ("a", " b", " c")
   .colle ct( Col lec tor s.t oCo lle cti on( Arr ayL ist ::n ew)); // creates new Arr
ayList
//Sorting with method references
List<String> names = List.of("Zoe", "Amy", "John");
List<S tri ng> sorted = names.s tr eam()
   .sorte d(S tri ng: :co mpa reT oIg nor eCase) // same as (a, b) -> a.comp are ToI g
no reC ase(b)
    .toList();
```

Concurrency terms

Concept	Analogy	Java Tool
Thread	A separate worker	Thread, Runnable, Callable
Race condition	Two people grabbing same sandwich	synchronized, locks, Atomic*
Thread pool	Team of workers managed by boss	ExecutorService
Results from a thread	Waiter bringing back your order	Future, Callable
Thread-safe collections		ConcurrentHashMap, CopyOnWriteArrayList

Concurrency: tasks appear to run at the same time, but may take turns sharing resources

Parallelism: Tasks actually run at the same time on different cores