

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. `FileNotFoundException` and `IOException` 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 `RuntimeException` or `Error`? then its a **checked exception**

Unchecked Exceptions

- Exceptional conditions that app cannot anticipate/recover from

- E.g. `NullPointerException` - 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 `RuntimeException`? then its an **unchecked exception**

Errors

Checked and unchecked exceptions and errors (cont)

E.g. `OutOfMemoryError` - 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 (direction) {
    case "left":
        goLeft();
        break;
    case "right":
        goRight();
        break;
    default:
        return "unknown";
}

// Java 14+
return switch (shirtNum) {
    case 1 -> "goalie";
    case 2 -> "left back";
    case 3, 4 -> "centre back";
    case 6, 7, 8 -> "midfielder";
    default -> throw new IllegalArgumentException("Invalid 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	<pre>List<String> list = Arrays.asList("apple", "banana", "cherry"); Stream<String> streamFromList = list.stream();</pre>
Creating a stream from an array	<pre>String[] array = {"apple", "banana", "cherry"}; Stream<String> streamFromArray = Arrays.stream(array);</pre>
Creating stream using Stream.of	<pre>Stream<String> streamOfElements = Stream.of("apple", "banana", "cherry");</pre>
Creating empty stream	<pre>Stream<String> emptyStream = Stream.empty();</pre>
Creating infinite streams	<pre>Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 1);</pre>

Primitive streams

mapToInt() vs map()

- `map(Streaming::length)` returns a `Stream<Integer>` (stream of Integer objects)

- `mapToInt(Streaming::length)` returns an `IntStream` (stream of primitive ints)

Specialised primitive streams

- `IntStream`, `LongStream`, `DoubleStream`

These streams have additional operations not available on regular streams:

- `sum: intStream.sum()`

- `average: intStream.average()`

- `statistics: intStream.summaryStatistics()`

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

Another stream ex.

```
List<Person> people = Arrays.asList(
    new Person("John", 25),
    new Person("Sarah", 32),
    new Person("Mike", 17),
    new Person("Emily", 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
<code>object::instanceMethod</code>	<code>x -> object.instanceMethod(x)</code>	Use object as the target for each call
<code>Class::staticMethod</code>	<code>x -> Class.staticMethod(x)</code>	Call a static method
<code>Class::instanceMethod</code>	<code>(obj, arg) -> obj.instanceMethod(arg)</code>	Useful in sorting or grouping

Concurrency: creating threads

```
// 1. by inheriting from Thread class
public class ExampleThread extends Thread {
    @Override // note override! invoked when thread starts
    public void run() { // we do NOT call RUN!!
        // contains all the code to execute when starting thread
        System.out.println(Thread.currentThread().getName());
    }
}
// start new thread..
public class ThreadExamples {
    public static void main(String[] args) {
        ExampleThread thread = new ExampleThread();
    }
}
```

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 icient Bal anc eEx ception
extends Exception {}
public class BankAc count {
    public void withdr aw( double amount)
throws Insuff icient Bal anc eEx ception {
        if (balance < amount) {
            throw new Insuff icient -
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 (Run ti 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<String> longFruits = fruits.stream()
    .filter(fruit -> fruit.length() > 5);
// map - transforms each element
Stream<String> fruitLengths = fruits.stream()
    .map(String::length);
// sorted - sorts elements
Stream<String> sortedFruits = fruits.stream()
    .sorted();
// distinct - removes duplicates
Stream<String> uniqueFruits = fruits.stream()
    .distinct();
// limit - reduces stream size
Stream<String> limitedFruits = fruits.stream()
    .limit(2);
// skip - skips elements
Stream<String> skipFruits = fruits.stream()
    .skip(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.stream().forEach(System.out::println);
// collect - puts each element into a collection
List<String> fruitList = fruits.stream()
    .filter(fruit -> fruit.length() > 5)
    .collect(Collectors.toList());
// reduce - reduces stream to single value
Optional<String> combined = fruits.stream()
    .reduce((a, b) -> a + ", " + b);
// count - returns number of elements
long count = fruits.stream().count();
// anyMatch/ allMatch/ noneMatch - check predicates
boolean anyLong = fruits.stream()
    .anyMatch(fruit -> fruit.length() > 5);
boolean allLong = fruits.stream()
    .allMatch(fruit -> fruit.length() > 3);
boolean noneLong = fruits.stream()
    .noneMatch(fruit -> fruit.length() > 10);
// findFirst/ findAny - find elements
Optional<String> first = fruits.stream().findFirst();
```

- 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 references, equivalent to x -> operation.equals(x) or x -> x.length.
- // **object :: instanceMethod**



By **nixik09**
cheatography.com/nixik09/

Not published yet.
Last updated 12th January, 2026.
Page 4 of 5.

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish
Yours!
<https://apollopapad.com>

Method references code samples (cont)

```
>String prefix = " Hel lo";

List<String> words = List.of("H ell o", " Hi", " Hey ");
boolean allMatch = words.stream()
    .allMatch(prefix::equals); // same: x -> prefix.equals(x)

//Class: :static Method
// Use for static utility methods, like from Math or Integer
List<String> numbers = List.of("1 ", " 2", " 3");
List<Integer> ints = numbers.stream()
    .map(Integer::parseInt) // same as: x -> Integer.parseInt(x)
    .toList();

List<Double> values = List.of(9.0, 16.0, 25.0);
List<Double> roots = values.stream()
    .map(Math::sqrt) // same as: x -> Math.sqrt(x)
    .toList();

//Class: :instance Method
List<String> items = List.of("H ell o", " Hi", " Hey ");
List<String> lower = items.stream()
    .map(String::toLowerCase) // same as: x -> x.toLowerCase()
    .toList();

List<String> nonEmpty = items.stream()
    .filter(s -> !s.isEmpty()); // classic
// or
List<String> nonEmpty2 = items.stream()
    .filter(Predicate.not(String::isEmpty)) // java 11+
    .toList();

//Constructor references - ClassName::new
List<String> list = Stream.of("a", " b", " c")
    .collect(Collectors.toCollection(ArrayList::new)); // creates new ArrayList

//Sorting with method references
List<String> names = List.of("Zoe", "Amy", "John");
List<String> sorted = names.stream()
    .sorted(String::compareToIgnoreCase) // same as (a, b) -> a.compareToIgnoreCase(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		



By [nixik09](#)
cheatography.com/nixik09/

Not published yet.
 Last updated 12th January, 2026.
 Page 5 of 5.

Sponsored by [ApolloPad.com](#)
 Everyone has a novel in them. Finish Yours!
<https://apollopadd.com>