edureka!

CORE JAVA CHEATSHEET

Learn **JAVA** from experts at https://www.edureka.co

Java Programming

Java is a high level, general purpose programming language that produces software for multiple platforms. It was developed by James Gosling in 1991 and released by Sun Microsystems in 1996 and is currently owned by Oracle.

Primitive Data Types

Type	Size	Range
byte	8	-128..127
short	16	-32,768..32,767
int	32	-2,147,483,648.. 2,147,483,647
long	64	9,223,372,036,854,775,808.. 9,223..
float	32	3.4e-0.38.. 3.4e+0.38
double	64	1.7e-308.. 1.7e+308
char	16	Complete Unicode Character Set
Boolean	1	True, False

Java Operators

Type	Operators
Arithmetic	+, – , *, ? , %
Assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=
Bitwise	^, &, |
Logical	&&, ||
Relational	<, >, <=, >=,==, !=
Shift	<<, >>, >>>
Ternary	? :
Unary	++x, –x, x++, x–, +x, –x, !, ~

Java Variables

```
{public|private} [static] type name [= expression|value];
```

Java Methods

```
{public|private} [static] {type | void} name(arg1, ..., argN ){statements}
```

Data Type Conversion

```
// Widening (byte<short<int<long<float<double)
int i = 10; //int--> long
long l = i; //automatic type conversion
// Narrowing
double d = 10.02;
long l = (long)d; //explicit type casting
// Numeric values to String
String str = String.valueOf(value);
// String to Numeric values
int i = Integer.parseInt(str);
double d = Double.parseDouble(str);
```

User Input

```
// Using BufferReader
BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
String name = reader.readLine();
// Using Scanner
Scanner in = new Scanner(System.in);
String s = in.nextLine();
int a = in.nextInt();
// Using Console
String name = System.console().readLine();
```

Iterative Statements

```
// for loop
for (condition) {expression}

// for each loop
for (int i: someArray) {}

// while loop
while (condition) {expression}

// do while loop
do {expression} while(condition)
```

Fibonacci series

```
for (i = 1; i <= n; ++i)
 {
   System.out.print(t1 + " + ");
   int sum = t1 + t2; t1 = t2;
   t2 = sum;
 }
```

Pyramid Pattern

```
k = 2*n - 2;
for(i=0; i<n; i++)
{
  for(j=0; j<k; j++){System.out.print(" ");}
  k = k - 1;
  for(j=0; j<=i; j++ ){System.out.print("* ");}
  System.out.println();
}
```

Decisive Statements

```
//if statement
if (condition) {expression}

//if-else statement
if (condition) {expression} else {expression}

//switch statement
switch (var) { case 1: expression; break;
default: expression; break; }
```

Prime Number

```
if (n < 2)
 {
   return false;
 }
for (int i=2; i <= n/i; i++)
 {
  if (n%i == 0) return false;
 }
return true;
```

Factorial of a Number

```
int factorial(int n)
{
   if (n == 0)
      {return 1;}
   else
      {
        return(n * factorial(n-1));
      }
}
```

Basic Java Program

```
public class Demo
 {
  public static void main(String[] args)
  {
    System.out.println("Hello from edureka!");
  }
 }
```

Save	className.java
Compile	javac className
Execute	java className

Arrays In Java

1 - Dimensional

```
// Initializing
type[] varName= new type[size];

// Declaring
type[] varName= new type[]{values1, value2,...};
```

Array with Random Variables

```
double[] arr = new double[n];
for (int i=0; i<n; i++)
{a[i] = Math.random();}
```

Maximum value in an Array

```
double max = 0;
for (int i=0; i<arr.length(); i++)
{  if(a[i] > max) max = a[i];  }
```

Reversing an Array

```
for(int i=0; i<(arr.length())/2; i++)
{ double temp = a[i];
  a[i] = a[n-1-i];
  a[n-1-i] = temp; }
```

Multi – Dimensional Arrays

```
// Initializing
datatype[][] varName  =  new dataType[row][col];
// Declaring
datatype[][] varName  =  {{value1, value2....},{value1,
value2....}..};
```

Transposing A Matrix

```
for(i = 0; i < row; i++)
 { for(j = 0; j < column; j++)
  { System.out.print(array[i][j]+" "); }
   System.out.println(" ");
 }
```

Multiplying two Matrices

```
for (i = 0; i < row1; i++)
 { for (j = 0; j < col2; j++)
  { for (k = 0; k < row2; k++)
   { sum = sum + first[i][k]*second[k][j]; }
 multiply[i][j] = sum;
 sum = 0;  } }
```

Java Strings

```
// Creating String using literal
String str1 = “Welcome”;

// Creating String using new keyword
String str2 = new String(”Edureka”);
```

String Methods

```
str1==str2 //compare the address;
String newStr = str1.equals(str2); //compares the values
String newStr = str1.equalsIgnoreCase() //
newStr = str1.length() //calculates length
newStr = str1.charAt(i) //extract i'th character
newStr = str1.toUpperCase() //returns string in ALL CAPS
newStr = str1.toLowerCase() //returns string in ALL LOWERCASE
newStr = str1.replace(oldVal, newVal) //search and replace
newStr = str1.trim() //trims surrounding whitespace
newStr = str1.contains("value"); //Check for the values
newStr = str1.toCharArray(); //Convert into character array
newStr = str1.IsEmpty(); //Check for empty String
newStr = str1.endsWith(); //Checks if string ends with the given suffix
```

SOLID

Single Responsibility Principle

> A class changes for only one reason

Open/Closed Principle

> A class should be open for extension, closed for editing

Liskov's Substitution Principle

> Derived types should cleanly and easily replace base types

Interface Segregation Principle

> Favor multiple single-purpose interfaces over composite

Dependency Inversion Principle

> Concrete classes depend on abstractions, not vice-versa

Other Principles

Don't Repeat Yourself (DRY)

> Duplication should be abstracted

Law of Demeter

> Only talk to related classes

Hollywood Principle

> "Don't call us, we'll call you"

You Ain't Gonna Need It

> Only code what you need now

Keep It Simple, Stupid

> Favor clarity over cleverness

Convention Over Configuration

> Defaults cover 90% of uses

Encapsulation

> What happens in Vegas...

Other Principles (cont)

Design By Contract

> And then write tests

Avoid Fragile Base Class

> Treat Base like a public API

Common Closure Principle

> Classes that change together, stay together

Common Refactorings

Encapsulate Field

Generalize Type

Type-Checking ⇒ State/Strategy

Conditional ⇒ Polymorphism

Extract Method

Extract Class

Move/Rename Method or Field

Move to Superclass/Subclass

http://martinfowler.com/refactoring/catalog

Class Associations: Association

Two objects have some sort of relationship to each other.
Example: *Car uses Highway*

Class Associations: Aggregation

An association where one object *has-a* (owns a) different object.
Example: *Car has a Driver*

Class Associations: Composition

An aggregation with dependency - objects are mutually destroyed/created.
Example: *Car has an Engine*

Class Associations: Generalization

"Is-A" relationship (inheritance).
Example: *Porsche is a Car*

Class Associations: Realization

One class implements behavior that is abstractly defined in another class.
Example: *An Animal may Move(), but a Duck would move by waddling*

Class Associations: Dependency

One class weakly depends on another.
Example: *Car uses Highway*

Access Modifiers

Private	Only inside the same class instance
Protected	Inside same or derived class instances
Public	All other classes linking/referencing the class
Internal	Only other classes in the same assembly
Protected Internal	All classes in same assembly, or derived classes in other assembly
Static	Accessible on the class itself (can combine with other accessors)

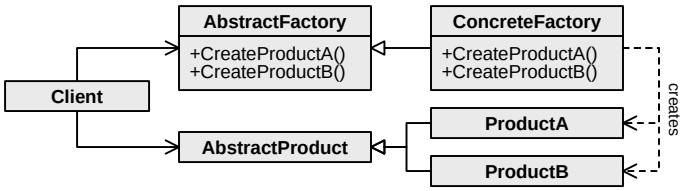
Design Patterns (GoF)	
Abstract Factory	Creational
Builder	Creational
Factory Method	Creational
Prototype	Creational
Singleton	Creational
Adapter	Structural
Bridge	Structural
Composite	Structural
Decorator	Structural
Facade	Structural
Flyweight	Structural
Proxy	Structural
Chain of Responsibility	Behavioral
Command	Behavioral
Interpreter	Behavioral
Iterator	Behavioral
Mediator	Behavioral
Memento	Behavioral
Observer	Behavioral
State	Behavioral
Strategy	Behavioral
Template Method	Behavioral
Visitor	Behavioral

Design Patterns Cheat Sheet

Creational Patterns

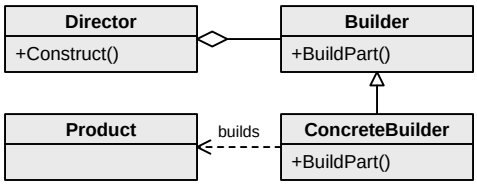
Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes



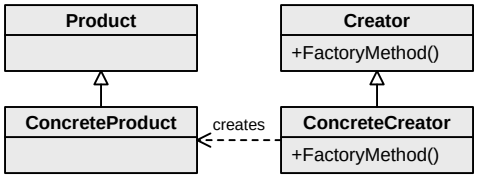
Builder

Separates the construction of a complex object from its representation so that the same construction process can create different representations.



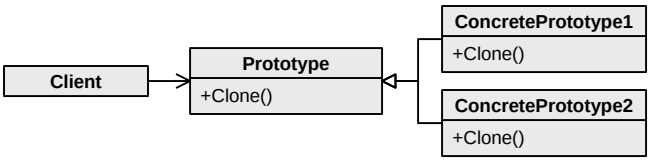
Factory Method

Defines an interface for creating an object but let subclasses decide which class to instantiate



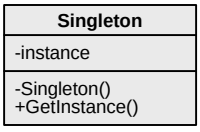
Prototype

Specifies the kinds of objects to create using a prototypical instance and create new objects by copying this prototype



Singleton

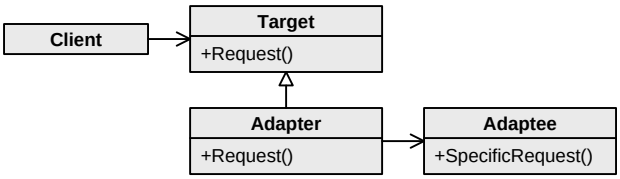
Ensure a class only has one instance and provide a global point of access to it



Structural Patterns

Adapter

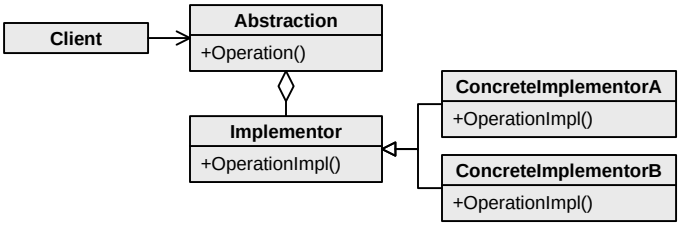
Converts the interface of a class into another interface clients expect



Structural Patterns (cont'd)

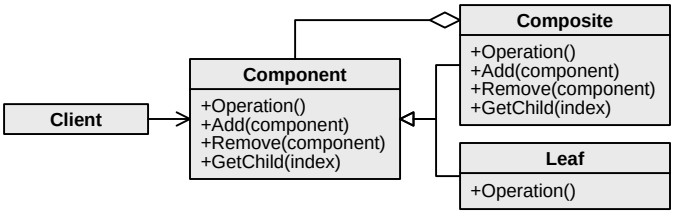
Bridge

Decouples an abstraction from its implementation so that the two can vary independently



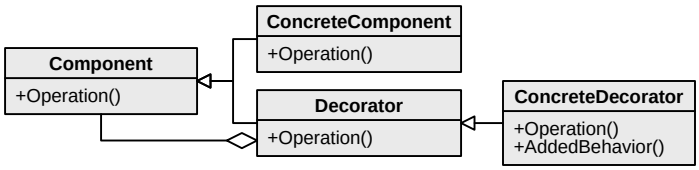
Composite

Composes objects into tree structures to represent part-whole hierarchies



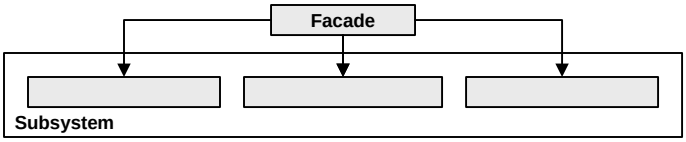
Decorator

Attaches additional responsibilities to an object dynamically



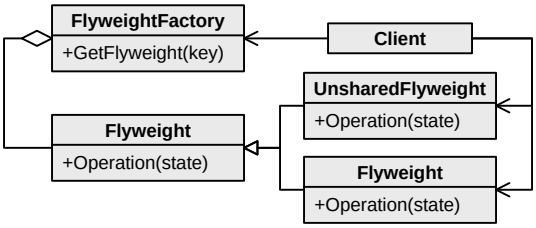
Facade

Provides a unified interface to a set of interfaces in a subsystem



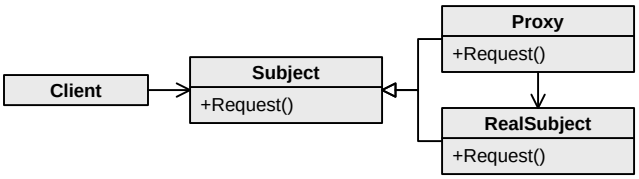
Flyweight

Uses sharing to support large numbers of fine-grained objects efficiently



Proxy

Provides a surrogate or placeholder for another object to control access to it

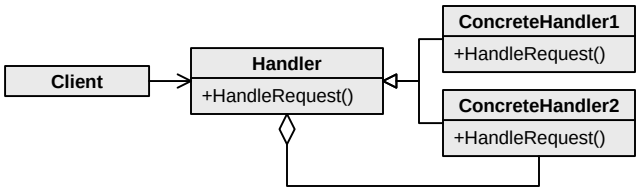


Design Patterns Cheat Sheet

Behavioral Patterns

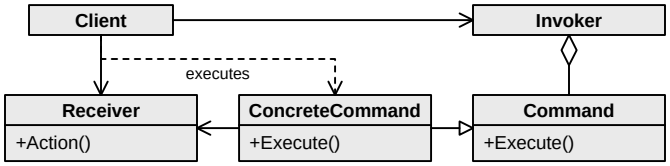
Chain of Responsibility

Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request



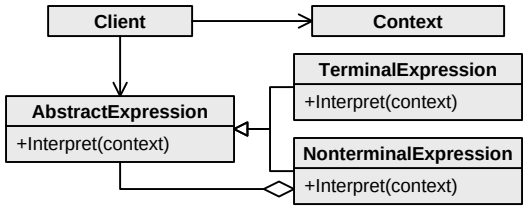
Command

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations



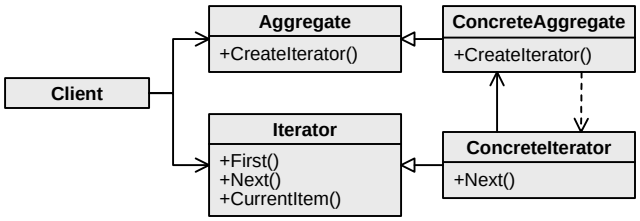
Interpreter

Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



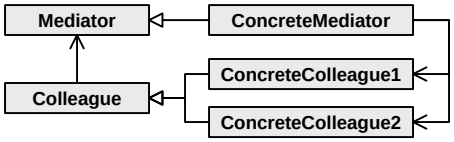
Iterator

Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



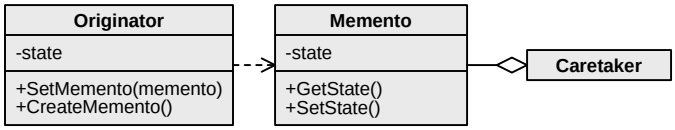
Mediator

Defines an object that encapsulates how a set of objects interact



Memento

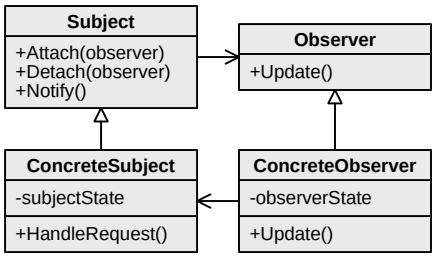
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later



Behavioral Patterns (cont'd)

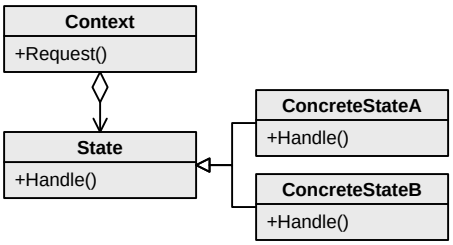
Observer

Defines a one-to-many dependency between objects so that when one object changes state all its dependents are notified and updated automatically



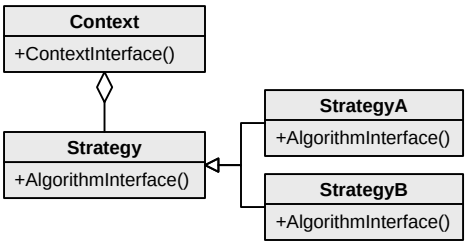
State

Allows an object to alter its behavior when its internal state changes



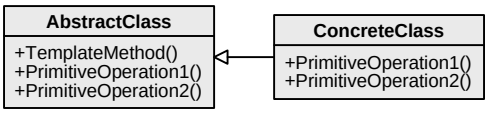
Strategy

Defines a family of algorithms, encapsulate each one, and make them interchangeable



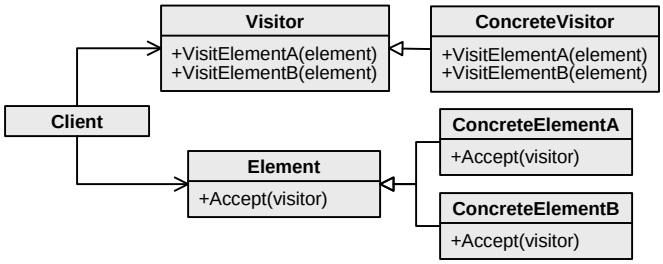
TemplateMethod

Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses



TemplateMethod

Represents an operation to be performed on the elements of an object structure



Java 8 Best Practices Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

BROUGHT TO YOU BY JRebel

Default methods

Evolve interfaces & create traits

```
//Default methods in interfaces
@FunctionalInterface
interface Utilities {
  default Consumer<Runnable> m() {
    return (r) -> r.run();
  }
  // default methods, still functional
  Object function(Object o);
}
class A implements Utilities { // implement
  public Object function(Object o) {
    return new Object();
  }
  {
    // call a default method
    Consumer<Runnable> n = new A().m();
  }
}
```

Lambdas

Syntax:
(parameters) -> expression
(parameters) -> { statements; }

```
// takes a Long, returns a String
Function<Long, String> f = (l) -> l.toString();
// takes nothing gives you Threads
Supplier<Thread> s = Thread::currentThread;
// takes a string as the parameter
Consumer<String> c = System.out::println;

// use them with streams
new ArrayList<String>().stream().
// peek: debug streams without changes
peek(e -> System.out.println(e)).
// map: convert every element into something
map(e -> e.hashCode()).
// filter: pass some elements through
filter (hc -> (hc % 2) == 0).
// collect all values from the stream
collect(Collectors.toCollection(TreeSet::new))
```

java.util. Optional

A container for possible null values

```
// Create an optional
Optional<String> optional =
Optional.ofNullable(a);
// process the optional
optional.map(s -> "RebelLabs:" + s);
// map a function that returns Optional
optional.flatMap(s -> Optional.ofNullable(s));

// run if the value is there
optional.ifPresent(System.out::println);
// get the value or throw an exception
optional.get();

// return the value or the given value
optional.orElse("Hello world!");
// return empty Optional if not satisfied
optional.filter(s -> s.startsWith("RebelLabs"));
```

Rules of Thumb

Traits: 1 default method per interface
Don't enhance functional interfaces
Only conservative implementations

Expressions over statements
Refactor to use method references
Chain lambdas rather than growing them

Fields - use plain objects
Method parameters, use plain objects
Return values - use Optional
Use *orElse()* instead of *get()*

Java Collections Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Notable Java collections libraries

Fastutil

http://fastutil.di.unimi.it/

Fast & compact type-specific collections for Java

Great default choice for collections of primitive types, like int or long. Also handles big collections with more than 2^{31} elements well.

Guava

https://github.com/google/guava

Google Core Libraries for Java 6+

Perhaps the default collection library for Java projects. Contains a magnitude of convenient methods for creating collection, like fluent builders, as well as advanced collection types.

Eclipse Collections

https://www.eclipse.org/collections/

Features you want with the collections you need

Previously known as gs-collections, this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.

JCTools

https://github.com/JCTools/JCTools

Java Concurrency Tools for the JVM.

If you work on high throughput concurrent applications and need a way to increase your performance, check out JCTools.

What can your collection do for you?

Collection class	Thread-safe alternative	Your data				Operations on your collections						
						Order of iteration				Random access		
		Individual elements	Key-value pairs	Duplicate element support	Primitive support	FIFO	Sorted	LIFO	Performant 'contains' check	By key	By value	By index
HashMap	ConcurrentHashMap	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashBiMap (Guava)	Maps.synchronizedBiMap (new HashBiMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
ArrayListMultimap (Guava)	Maps.synchronizedMultiMap (new ArrayListMultimap())	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
LinkedHashMap	Collections.synchronizedMap (new LinkedHashMap())	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗
TreeMap	ConcurrentSkipListMap	✗	✓	✗	✗	✗	✓	✗	✓*	✓*	✗	✗
Int2IntMap (Fastutil)		✗	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
ArrayList	CopyOnWriteArrayList	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
HashSet	Collections.newSetFromMap (new ConcurrentHashMap<>())	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
IntArrayList (Fastutil)		✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓
PriorityQueue	PriorityBlockingQueue	✓	✗	✓	✗	✗	✓**	✗	✗	✗	✗	✗
ArrayDeque	ArrayBlockingQueue	✓	✗	✓	✗	✓**	✗	✓**	✗	✗	✗	✗

** O(log(n)) complexity, while all others are O(1) - constant time*

*** when using Queue interface methods: offer() / poll()*

How fast are your collections?

Collection class	Random access by index / key	Search / Contains	Insert
ArrayList	O(1)	O(n)	O(n)
HashSet	O(1)	O(1)	O(1)
HashMap	O(1)	O(1)	O(1)
TreeMap	O(log(n))	O(log(n))	O(log(n))

Remember, not all operations are equally fast. Here's a reminder of how to treat the Big-O complexity notation:

O(1) - constant time, really fast, doesn't depend on the size of your collection

O(log(n)) - pretty fast, your collection size has to be extreme to notice a performance impact

O(n) - linear to your collection size: the larger your collection is, the slower your operations will be

BROUGHT TO YOU BY JRebel

Java Generics cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Basics

Generics don't exist at runtime!

```
class Pair<T1, T2> { /* ... */ }
```
-- the type parameter section, in angle brackets, specifies type variables.

Type parameters are substituted when objects are instantiated.

```
Pair<String, Long> p1 = new
  Pair<String, Long> ("RL", 43L);
```

Avoid verbosity with the diamond operator:

```
Pair<String, Long> p1 =
  new Pair<>("RL", 43L);
```

Wildcards

`Collection<Object>` - heterogenous, any object goes in.
`Collection<?>` - homogenous collection of arbitrary type.

Avoid using wildcards in return types!

Intersection types

```
<T extends Object &
  Comparable<? super T>> T
max(Collection<? extends T> coll)
```

The return type here is ***Object***!

Compiler generates the bytecode for the most general method only.

Producer Extends Consumer Super (PECS)

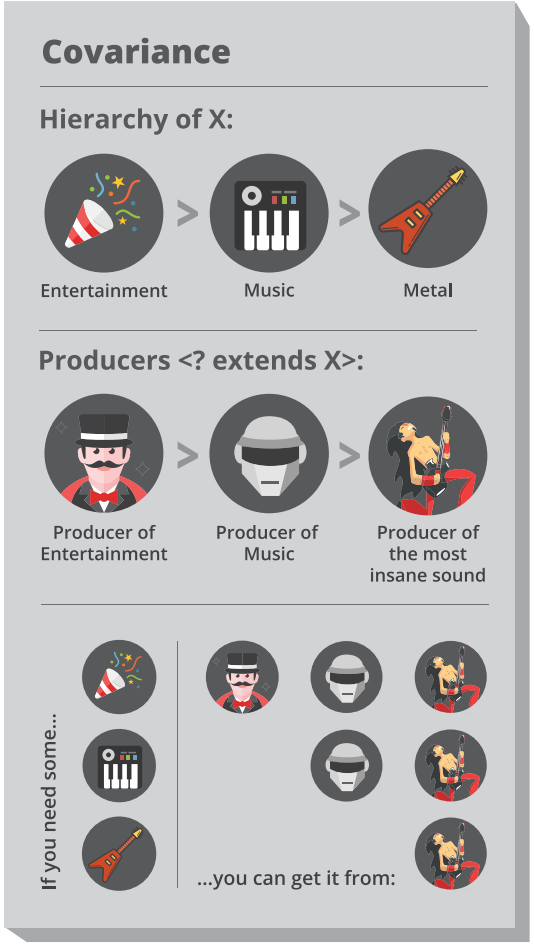
```
Collections.copy(List<? super T> dest, List<? extends T> src)
```

src -- contains elements of type T or its subtypes.

dest -- accepts elements, so defined to use T or its supertypes.

*Consumers are **contravariant** (use super).*

*Producers are **covariant** (use extends).*



Method Overloading

```
String f(Object s) {
  return "object";
}
String f(String s) {
  return "string";
}
<T> String generic(T t) {
  return f(t);
}
```

If called **generic("string")** returns "object".

Recursive generics

Recursive generics add constraints to your type variables. This helps the compiler to better understand your types and API.

```
interface Cloneable<T extends
Cloneable<T>> {
  T clone();
}
```

Now `cloneable.clone().clone()` will compile.

Covariance

```
List<Number> ⊁ ArrayList<Integer>
```

Collections are not covariant!

BROUGHT TO YOU BY JRebel

Java 8 Streams Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Definitions

- ☑ A stream **is** a pipeline of functions that can be evaluated.
- ☑ Streams **can** transform data.
- ☒ A stream **is not** a data structure.
- ☒ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
     .map(book -> book.getAuthor())
     .filter(author -> author.getAge() >= 50)
     .distinct()
     .limit(15)
     .map(Author::getSurname)
     .map(String::toUpperCase)
     .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
     .map(Book::getAuthor)
     .filter(a -> a.getGender() == Gender.FEMALE)
     .map(Author::getAge)
     .filter(age -> age < 25)
     .reduce(0, Integer::sum):
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
     groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
     .map(member -> member.getFollowers())
     .flatMap(followers -> followers.stream())
     .collect(toList());
```

Pitfalls

☒ Don't update shared mutable variables i.e.

```
List<Book> myList =
new ArrayList<>();
library.stream().forEach
(e -> myList.add(e));
```

☒ Avoid blocking operations when using parallel streams.

BROUGHT TO YOU BY JRebel

Git Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new_branch

```
$ git branch new_branch
```

Delete the branch called my_branch

```
$ git branch -d my_branch
```

Merge branch_a into branch_b

```
$ git checkout branch_b
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

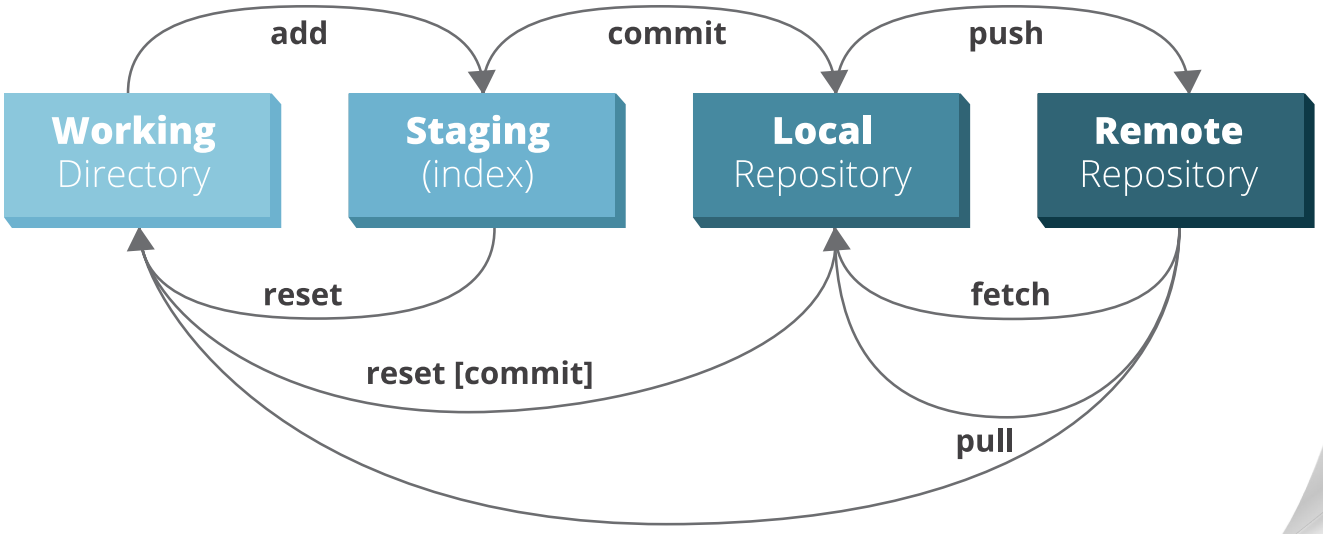
```
$ git push
```

Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit https://training.github.com/ for official GitHub training.



BROUGHT TO YOU BY JRebel

Maven cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Getting started with Maven

Create Java project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
```

Create web project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
-DarchetypeArtifactId=maven-archetype-webapp
```

Create archetype from existing project

```
mvn archetype:create-from-project
```

Main phases

clean — delete target directory
validate — validate, if the project is correct
compile — compile source code, classes stored in target/classes
test — run tests
package — take the compiled code and package it in its distributable format, e.g. JAR, WAR
verify — run any checks to verify the package is valid and meets quality criteria
install — install the package into the local repository
deploy — copies the final package to the remote repository

Useful command line options

-DskipTests=true compiles the tests, but skips running them

-Dmaven.test.skip=true skips compiling the tests and does not run them

-T - number of threads:
-T 4 is a decent default
-T 2C - 2 threads per CPU

-rf, --resume-from resume build from the specified project

-pl, --projects makes Maven build only specified modules and not the whole project

-am, --also-make makes Maven figure out what modules out target depends on and build them too

-o, --offline work offline

-X, --debug enable debug output

-P, --activate-profiles comma-delimited list of profiles to activate

-U, --update-snapshots forces a check for updated dependencies on remote repositories

-ff, --fail-fast stop at first failure

Essential plugins

Help plugin — used to get relative information about a project or the system.
mvn help:describe describes the attributes of a plugin
mvn help:effective-pom displays the effective POM as an XML for the current build, with the active profiles factored in.

Dependency plugin — provides the capability to manipulate artifacts.
mvn dependency:analyze analyzes the dependencies of this project
mvn dependency:tree prints a tree of dependencies

Compiler plugin — compiles your java code.
Set language level with the following configuration:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</
  artifactId>
 <version>3.6.1</version>
 <configuration>
   <source>1.8</source>
   <target>1.8</target>
 </configuration>
</plugin>
```

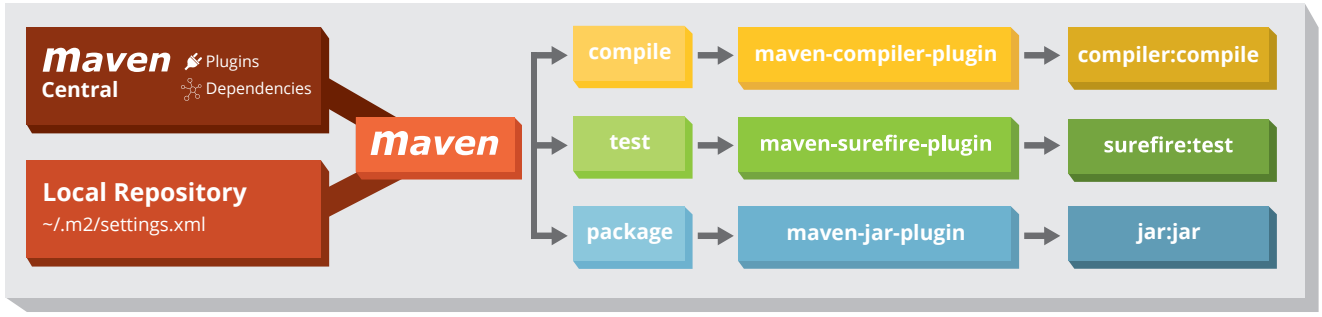
Version plugin — used when you want to manage the versions of artifacts in a project's POM.

Wrapper plugin — an easy way to ensure a user of your Maven build has everything that is necessary.

Spring Boot plugin — compiles your Spring Boot app, build an executable fat jar.

Exec — amazing general purpose plugin,

The big picture



BROUGHT TO YOU BY JRebel

SQL cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Basic Queries

```
-- filter your columns
   SELECT col1, col2, col3, ... FROM table1
-- filter the rows
   WHERE col4 = 1 AND col5 = 2
-- aggregate the data
   GROUP by ...
-- limit aggregated data
   HAVING count(*) > 1
-- order of the results
   ORDER BY col2
```

Useful keywords for **SELECTS**:

DISTINCT - return unique results
BETWEEN a **AND** b - limit the range, the values can be numbers, text, or dates
LIKE - pattern search within the column text
IN (a, b, c) - check if the value is contained among given.

Data Modification

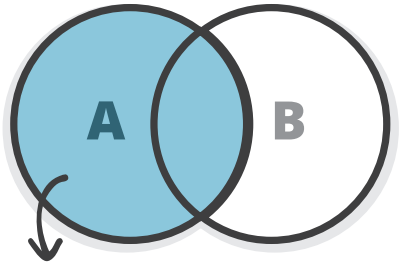
```
-- update specific data with the WHERE clause
   UPDATE table1 SET col1 = 1 WHERE col2 = 2
-- insert values manually
   INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME)
      VALUES (1, 'Rebel', 'Labs');
-- or by using the results of a query
   INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME)
      SELECT id, last_name, first_name FROM table2
```

Views

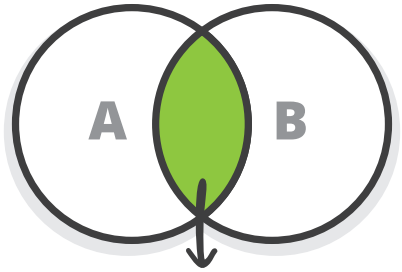
A **VIEW** is a virtual table, which is a result of a query.
They can be used to create virtual tables of complex queries.

```
CREATE VIEW view1 AS
SELECT col1, col2
FROM table1
WHERE ...
```

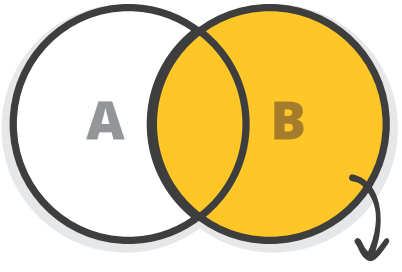
The Joy of JOINs



LEFT OUTER JOIN - *all rows from table A, even if they do not exist in table B*



INNER JOIN - *fetch the results that exist in both tables*



RIGHT OUTER JOIN - *all rows from table B, even if they do not exist in table A*

Updates on JOINed Queries

You can use **JOIN**s in your **UPDATE**s

```
UPDATE t1 SET a = 1
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id
WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

NB! Use database specific syntax, it might be faster!

Semi JOINs

You can use subqueries instead of **JOIN**s:

```
SELECT col1, col2 FROM table1 WHERE id IN
   (SELECT t1_id FROM table2 WHERE date >
      CURRENT_TIMESTAMP)
```

Indexes

If you query by a column, index it!

```
CREATE INDEX index1 ON table1 (col1)
```

Don't forget:

Avoid overlapping indexes
Avoid indexing on too many columns
Indexes can speed up **DELETE** and **UPDATE** operations

Useful Utility Functions

```
-- convert strings to dates:
   TO_DATE (Oracle, PostgreSQL), STR_TO_DATE (MySQL)
-- return the first non-NULL argument:
   COALESCE (col1, col2, "default value")
-- return current time:
   CURRENT_TIMESTAMP
-- compute set operations on two result sets
   SELECT col1, col2 FROM table1
   UNION / EXCEPT / INTERSECT
   SELECT col3, col4 FROM table2;
```

Union - returns data from both queries
Except - rows from the first query that are not present in the second query
Intersect - rows that are returned from both queries

Reporting

Use aggregation functions

COUNT - return the number of rows
SUM - cumulate the values
AVG - return the average for the group
MIN / MAX - smallest / largest value

BROUGHT TO YOU BY XRebel

Regex cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Character classes

`[abc]` matches **a** or **b**, or **c**.
`[^abc]` negation, matches everything except **a**, **b**, or **c**.
`[a-c]` range, matches **a** or **b**, or **c**.
`[a-c[f-h]]` union, matches **a**, **b**, **c**, **f**, **g**, **h**.
`[a-c&&[b-c]]` intersection, matches **b** or **c**.
`[a-c&&[^b-c]]` subtraction, matches **a**.

Predefined character classes

`.` Any character.
`\d` A digit: `[0-9]`
`\D` A non-digit: `[^0-9]`
`\s` A whitespace character: `[ \t\n\x0B\f\r]`
`\S` A non-whitespace character: `[^\s]`
`\w` A word character: `[a-zA-Z_0-9]`
`\W` A non-word character: `[^\w]`

Boundary matches

`^` The beginning of a line.
`$` The end of a line.
`\b` A word boundary.
`\B` A non-word boundary.
`\A` The beginning of the input.
`\G` The end of the previous match.
`\Z` The end of the input but for the final terminator, if any.
`\z` The end of the input.

Pattern flags

`Pattern.CASE_INSENSITIVE` - enables case-insensitive matching.
`Pattern.COMMENTS` - whitespace and comments starting with **#** are ignored until the end of a line.
`Pattern.MULTILINE` - one expression can match multiple lines.
`Pattern.UNIX_LINES` - only the '**\n**' line terminator is recognized in the behavior of **.**, **^**, and **$**.

Useful Java classes & methods

PATTERN

A pattern is a compiler representation of a regular expression.

`Pattern compile(String regex)`
Compiles the given regular expression into a pattern.

`Pattern compile(String regex, int flags)`
Compiles the given regular expression into a pattern with the given flags.

`boolean matches(String regex)`
Tells whether or not this string matches the given regular expression.

`String[] split(CharSequence input)`
Splits the given input sequence around matches of this pattern.

`String quote(String s)`
Returns a literal pattern String for the specified String.

`Predicate<String> asPredicate()`
Creates a predicate which can be used to match a string.

MATCHER

An engine that performs match operations on a character sequence by interpreting a Pattern.

`boolean matches()`
Attempts to match the entire region against the pattern.

`boolean find()`
Attempts to find the next subsequence of the input sequence that matches the pattern.

`int start()`
Returns the start index of the previous match.

`int end()`
Returns the offset after the last character matched.

Quantifiers

Greedy	Reluctant	Possessive	Description
X?	X??	X?+	*X, once or not at all.*
X*	X*?	X*+	*X, zero or more times.*
X+	X+?	X++	*X, one or more times.*
X{n}	X{n}?	X{n}+	*X, exactly n times.*
X{n,}	X{n,}?	X{n,}+	*X, at least n times.*
X{n,m}	X{n,m}?	X{n,m}+	*X, at least n but not more than m times.*

Greedy - matches the longest matching group.
Reluctant - matches the shortest group.
Possessive - longest match or bust (no backoff).

Groups & backreferences

A group is a captured subsequence of characters which may be used later in the expression with a backreference.

`(...)` - defines a group.
`\N` - refers to a matched group.

`(\d\d)` - a group of two digits.
`(\d\d)/\1`- two digits repeated twice.
`\1` - refers to the matched group.

Logical operations

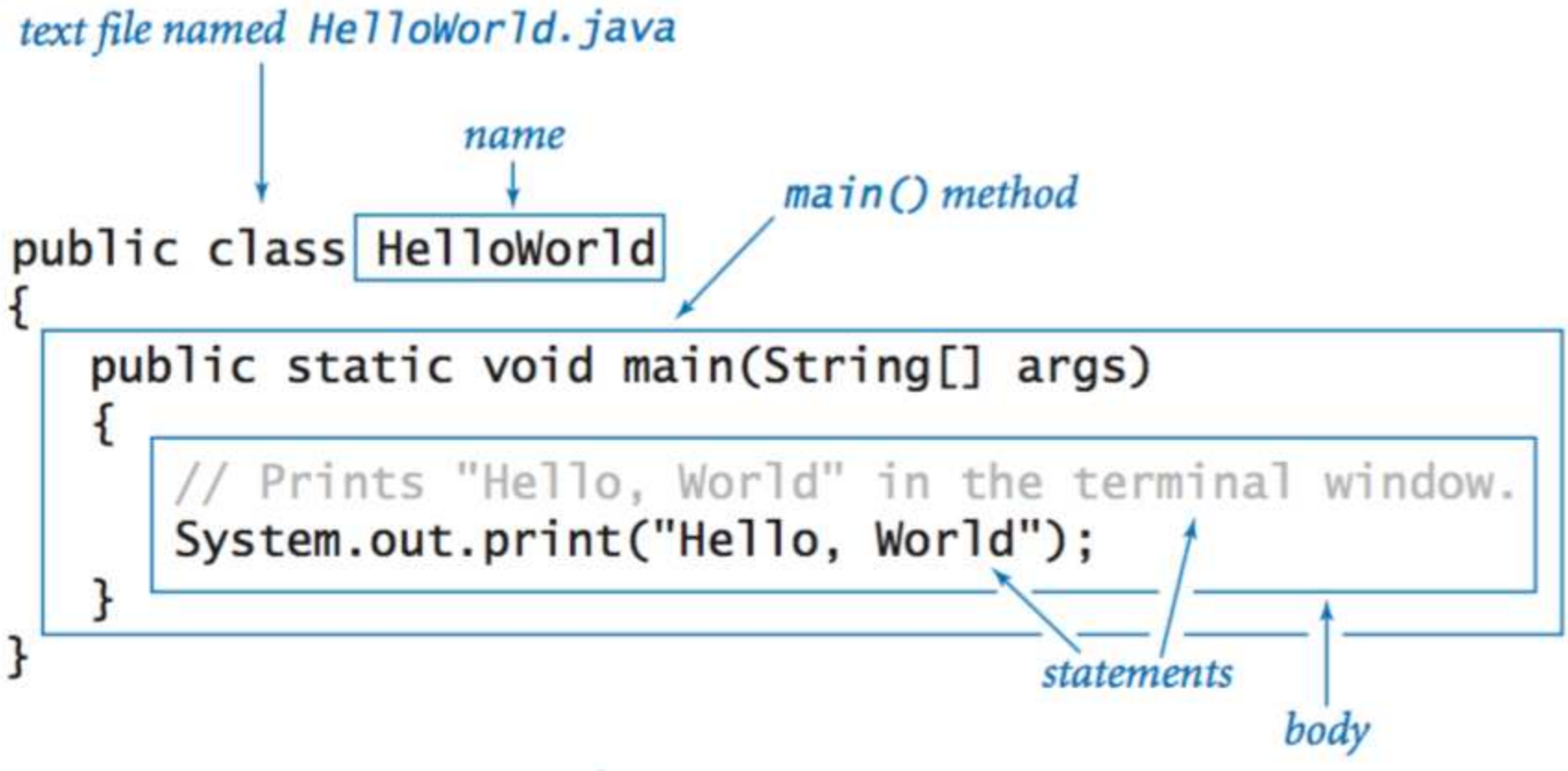
XY **X** then **Y**.
X|Y **X** or **Y**.

BROUGHT TO YOU BY JRebel

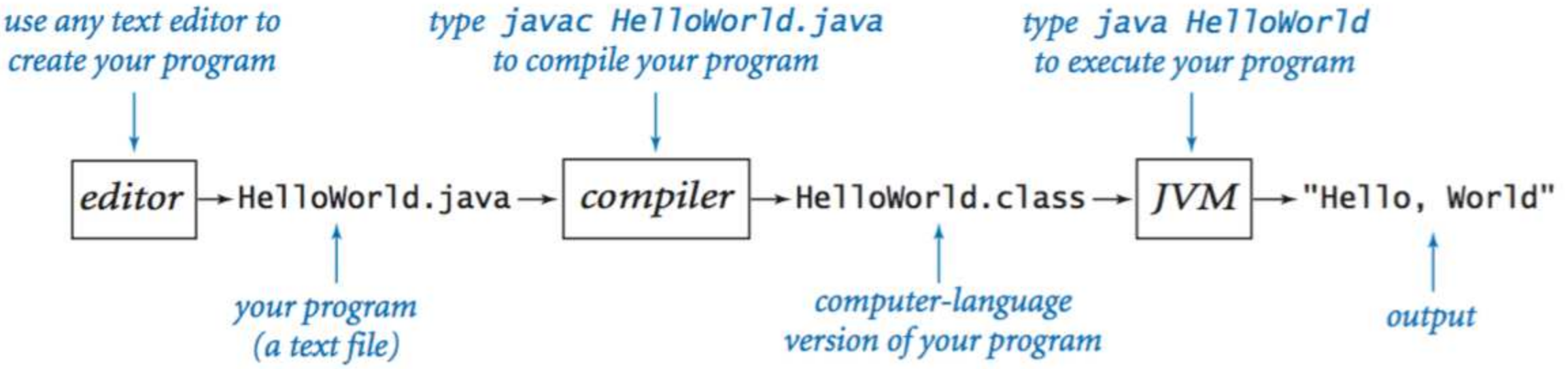
Java Programming Cheatsheet

We summarize the most commonly used Java language features and APIs in the textbook.

Hello, World.



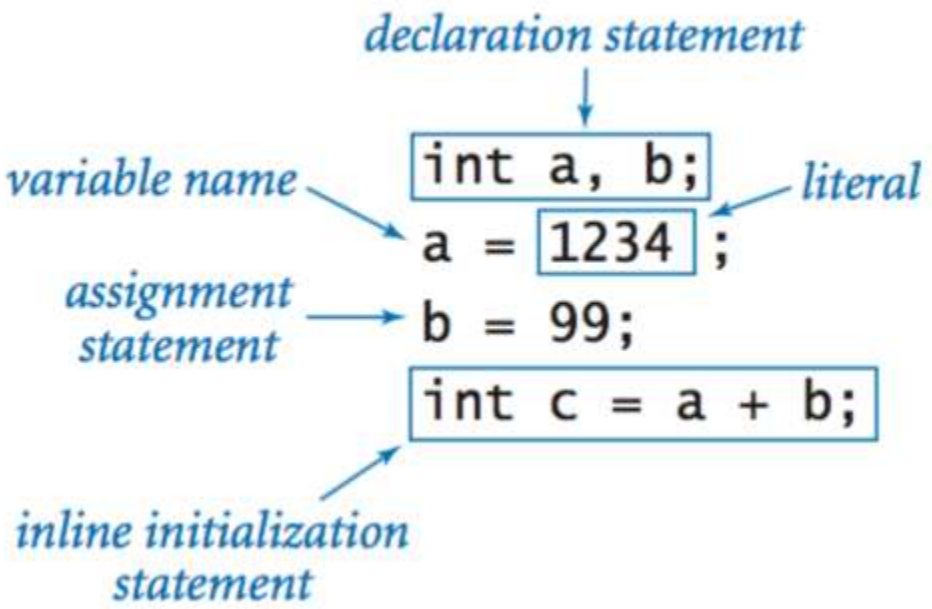
Editing, compiling, and executing.



Built-in data types.

type	*set of values*	*common operators*	*sample literal values*
`int`	integers	`+ - * / %`	`99 12 2147483647`
`double`	floating-point numbers	`+ - * /`	`3.14 2.5 6.022e23`
`boolean`	boolean values	`&& || !`	`true false`
`char`	characters		`'A' '1' '%' '\n'`
`String`	sequences of characters	`+`	`"AB" "Hello" "2.5"`

Declaration and assignment statements.



Integers.

values	integers between -2^{31} and $+2^{31}-1$					
typical literals			1234 99 0 1000000			
operations	*sign*	*add*	*subtract*	*multiply*	*divide*	*remainder*
operators	`+ -`	`+`	`-`	`*`	`/`	`%`

expression	*value*	*comment*
`99`	`99`	*integer literal*
`+99`	`99`	*positive sign*
`-99`	`-99`	*negative sign*
`5 + 3`	`8`	*addition*
`5 - 3`	`2`	*subtraction*
`5 * 3`	`15`	*multiplication*
`5 / 3`	`1`	*no fractional part*
`5 % 3`	`2`	*remainder*
`1 / 0`		*run-time error*
`3 * 5 - 2`	`13`	** has precedence*
`3 + 5 / 2`	`5`	*/ has precedence*
`3 - 5 - 2`	`-4`	*left associative*
`( 3 - 5 ) - 2`	`-4`	*better style*
`3 - ( 5 - 2 )`	`0`	*unambiguous*

Floating-point numbers.

values	real numbers (specified by IEEE 754 standard)			
typical literals	`3.14159`	`6.022e23`	`2.0`	`1.4142135623730951`
operations	*add*	*subtract*	*multiply*	*divide*
operators	`+`	`-`	`*`	`/`

expression	*value*
`3.141 + 2.0`	`5.141`
`3.141 - 2.0`	`1.111`
`3.141 / 2.0`	`1.5705`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Booleans.

values	*true or false*
literals	`true` `false`
operations	and or not
operators	&& || !

a	*!a*
`true`	`false`
`false`	`true`

a	*b*	*a && b*	*a || b*
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Comparison operators.

op	*meaning*	*true*	*false*
==	*equal*	`2 == 2`	`2 == 3`
!=	*not equal*	`3 != 2`	`2 != 2`
<	*less than*	`2 < 13`	`2 < 2`
<=	*less than or equal*	`2 <= 2`	`3 <= 2`
>	*greater than*	`13 > 2`	`2 > 13`
>=	*greater than or equal*	`3 >= 2`	`2 >= 3`

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Printing.

`void System.out.print(String s)`	*print s*
`void System.out.println(String s)`	*print s, followed by a newline*
`void System.out.println()`	*print a newline*

Parsing command-line arguments.

int	Integer.parseInt(String s)	*convert* s *to an* int *value*
double	Double.parseDouble(String s)	*convert* s *to a* double *value*
long	Long.parseLong(String s)	*convert* s *to a* long *value*

Math library.

public class Math

double	abs(double a)	*absolute value of* a
double	max(double a, double b)	*maximum of* a *and* b
double	min(double a, double b)	*minimum of* a *and* b
double	sin(double theta)	*sine of* theta
double	cos(double theta)	*cosine of* theta
double	tan(double theta)	*tangent of* theta
double	toRadians(double degrees)	*convert angle from degrees to radians*
double	toDegrees(double radians)	*convert angle from radians to degrees*
double	exp(double a)	*exponential* (e^a)
double	log(double a)	*natural log* ($\log_e a$, *or ln* a)
double	pow(double a, double b)	*raise* a *to the* b*th power* (a^b)
long	round(double a)	*round* a *to the nearest integer*
double	random()	*random number in* [0, 1)
double	sqrt(double a)	*square root of* a
double	E	*value of* e *(constant)*
double	PI	*value of* π *(constant)*

The full java.lang.Math API.

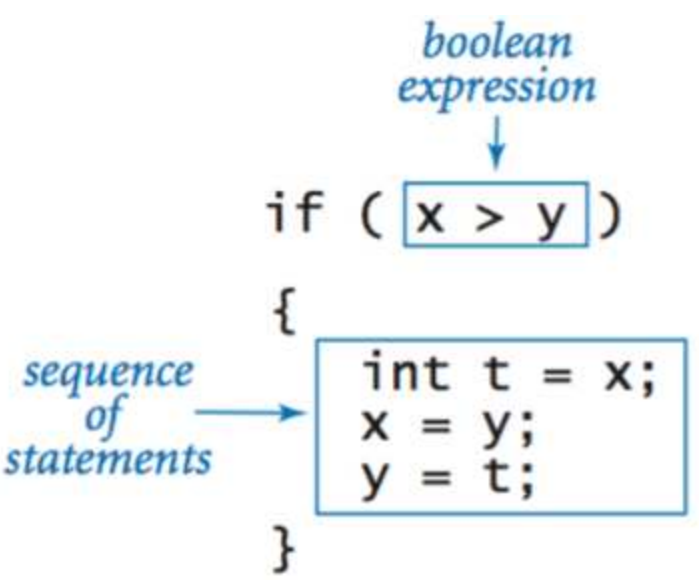
Java library calls.

method call	*library*	*return type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Double.parseDouble("1.5")`	`Double`	`double`	`1.5`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.log(Math.E)`	`Math`	`double`	`1.0`
`Math.random()`	`Math`	`double`	*random in* [0, 1)
`Math.round(3.14159)`	`Math`	`long`	`3`
`Math.max(1.0, 9.0)`	`Math`	`double`	`9.0`

Type conversion.

expression	*expression type*	*expression value*
`(1 + 2 + 3 + 4) / 4.0`	`double`	`2.5`
`Math.sqrt(4)`	`double`	`2.0`
`"1234" + 99`	`String`	`"123499"`
`11 * 0.25`	`double`	`2.75`
`(int) 11 * 0.25`	`double`	`2.75`
`11 * (int) 0.25`	`int`	`0`
`(int) (11 * 0.25)`	`int`	`2`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`Integer.parseInt("1234")`	`int`	`1234`

Anatomy of an if statement.



If and if-else statements.

absolute value

```
if (x < 0) x = -x;
```

put the smaller value in x and the larger value in y

```
if (x > y)
{
   int t = x;
   x = y;
   y = t;
}
```

maximum of x and y

```
if (x > y) max = x;
else       max = y;
```

error check for division operation

```
if (den == 0) System.out.println("Division by zero");
else          System.out.println("Quotient = " + num/den);
```

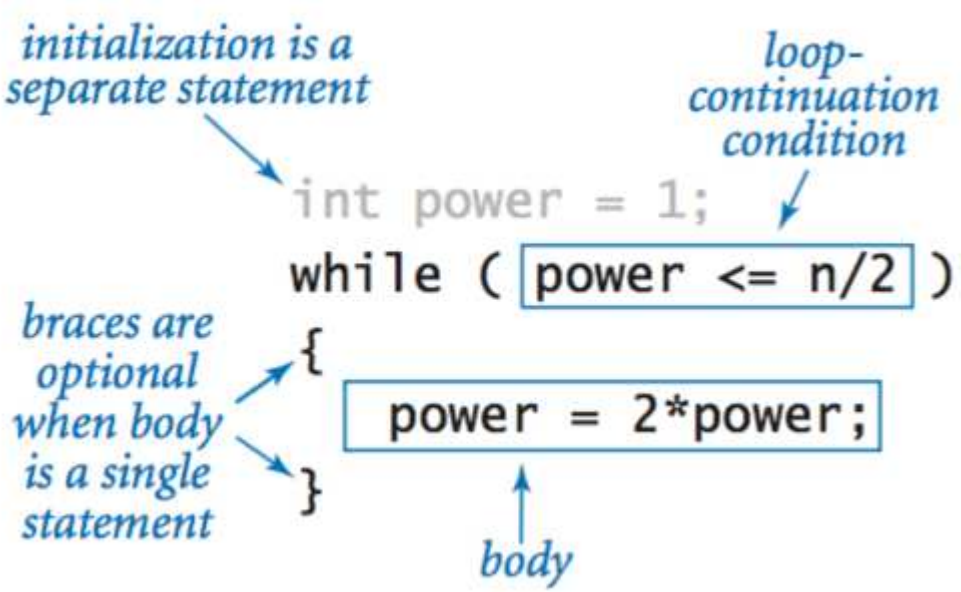
error check for quadratic formula

```
double discriminant = b*b - 4.0*c;
if (discriminant < 0.0)
{
   System.out.println("No real roots");
}
else
{
   System.out.println((-b + Math.sqrt(discriminant))/2.0);
   System.out.println((-b - Math.sqrt(discriminant))/2.0);
}
```

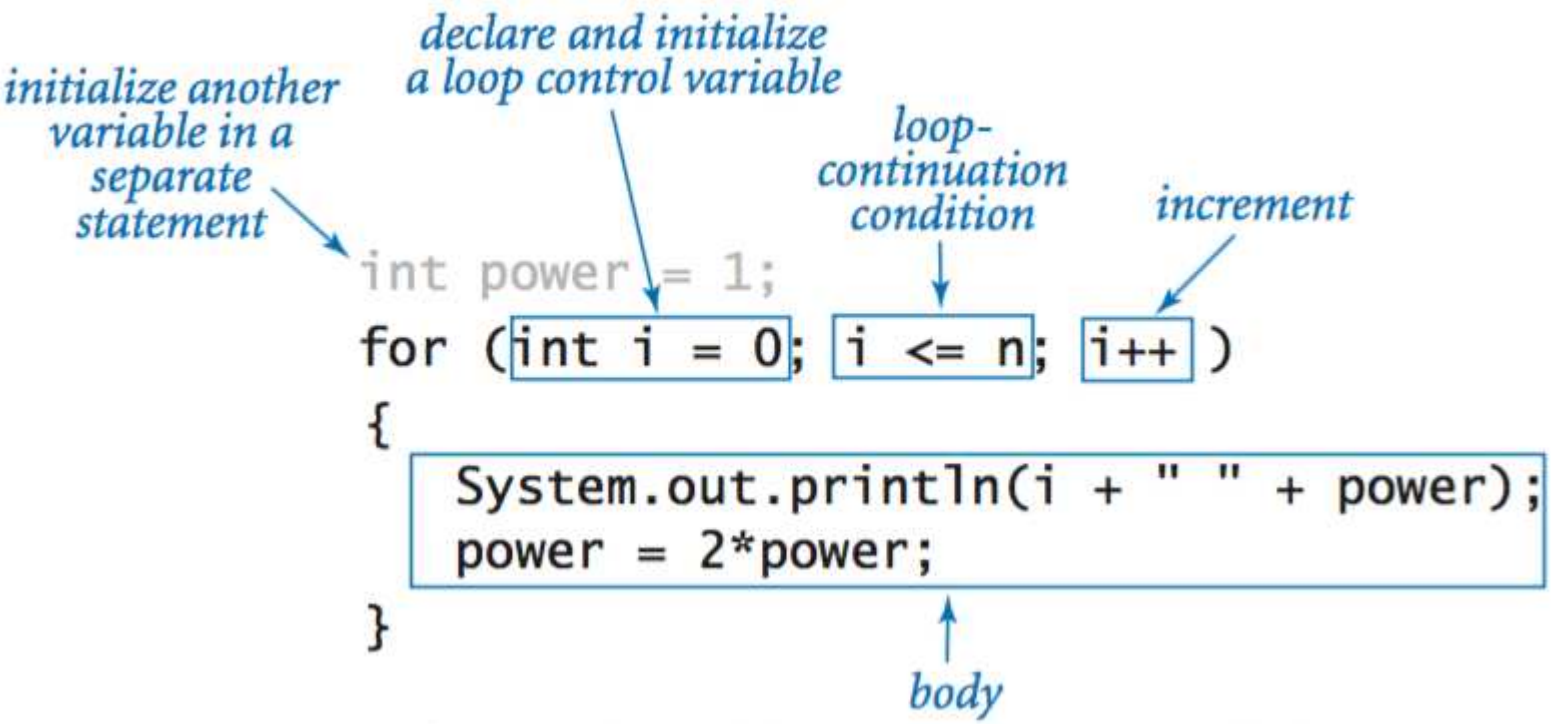
Nested if-else statement.

```
if      (income <      0) rate = 0.00;
else if (income <   8925) rate = 0.10;
else if (income <  36250) rate = 0.15;
else if (income <  87850) rate = 0.23;
else if (income < 183250) rate = 0.28;
else if (income < 398350) rate = 0.33;
else if (income < 400000) rate = 0.35;
else                      rate = 0.396;
```

Anatomy of a while loop.



Anatomy of a for loop.



Loops.

compute the largest power of 2 less than or equal to n	`int power = 1;`
	`while (power <= n/2)`
	`  power = 2*power;`
	`System.out.println(power);`

compute a finite sum (1 + 2 + ... + n)	`int sum = 0;`
	`for (int i = 1; i <= n; i++)`
	`  sum += i;`
	`System.out.println(sum);`

compute a finite product (n! = 1 × 2 × ... × n)	`int product = 1;`
	`for (int i = 1; i <= n; i++)`
	`  product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0; i <= n; i++)` `  System.out.println(i + " " + 2*Math.PI*i/n);`

compute the ruler function (see PROGRAM 1.2.1)	`String ruler = "1";`
	`for (int i = 2; i <= n; i++)`
	`  ruler = ruler + " " + i + " " + ruler;`
	`System.out.println(ruler);`

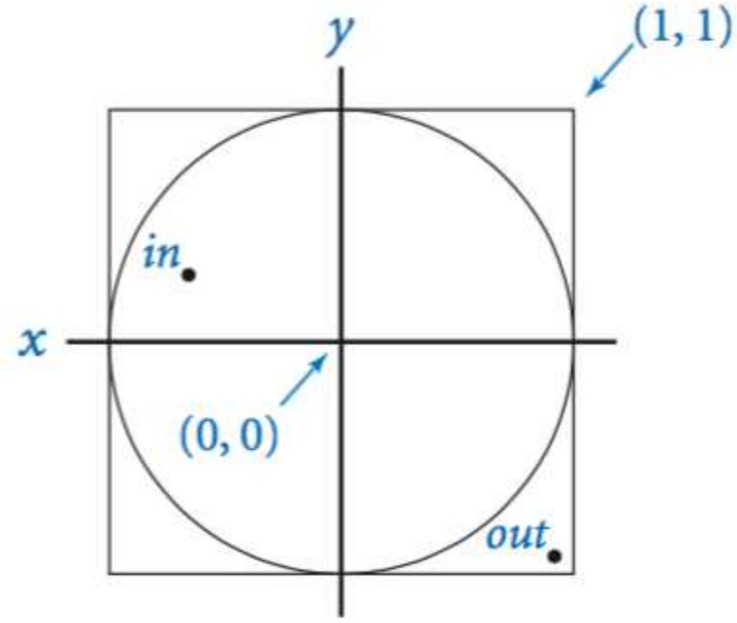
Break statement.

```
int factor;
for (factor = 2; factor <= n/factor; factor++)
   if (n % factor == 0) break;

if (factor > n/factor)
   System.out.println(n + " is prime");
```

Do-while loop.

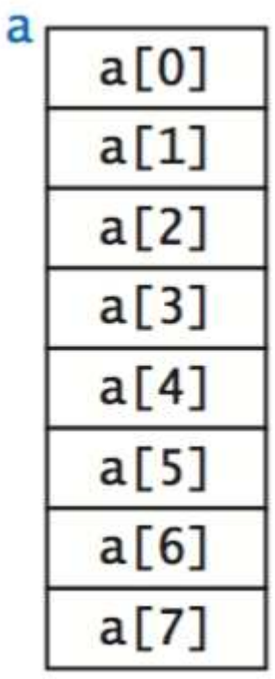
```
do
{  // Scale x and y to be random in (-1, 1).
   x = 2.0*Math.random() - 1.0;
   y = 2.0*Math.random() - 1.0;
} while (Math.sqrt(x*x + y*y) > 1.0);
```



Switch statement.

```
switch (day) {
   case 0: System.out.println("Sun"); break;
   case 1: System.out.println("Mon"); break;
   case 2: System.out.println("Tue"); break;
   case 3: System.out.println("Wed"); break;
   case 4: System.out.println("Thu"); break;
   case 5: System.out.println("Fri"); break;
   case 6: System.out.println("Sat"); break;
}
```

Arrays.



Inline array initialization.

```
String[] SUITS = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] RANKS = {
   "2", "3", "4", "5", "6", "7", "8", "9", "10",
   "Jack", "Queen", "King", "Ace"
};
```

Typical array-processing code.

create an array with random values	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

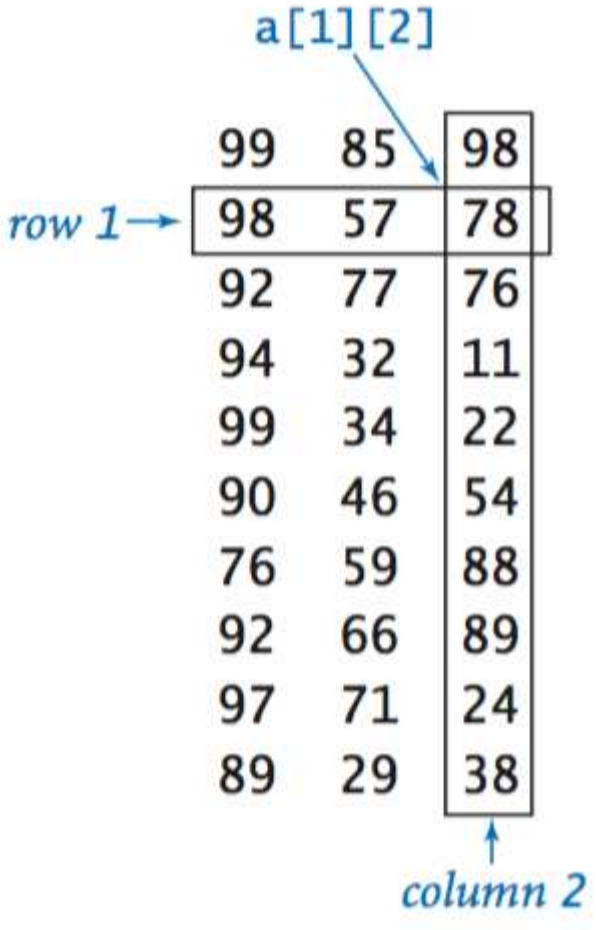
find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`  sum += a[i];`
	`double average = sum / n;`

reverse the values within an array	`for (int i = 0; i < n/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[n-1-i];`
	`  a[n-i-1] = temp;`
	`}`

copy sequence of values to another array	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

Two-dimensional arrays.



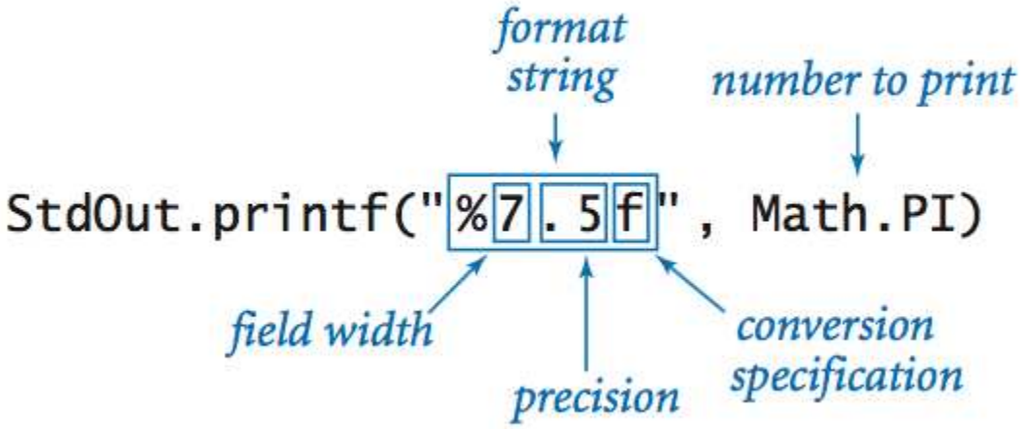
Inline initialization.

```
double [][] a =
{
   { 99.0, 85.0, 98.0,  0.0 },
   { 98.0, 57.0, 79.0,  0.0 },
   { 92.0, 77.0, 74.0,  0.0 },
   { 94.0, 62.0, 81.0,  0.0 },
   { 99.0, 94.0, 92.0,  0.0 },
   { 80.0, 76.5, 67.0,  0.0 },
   { 76.0, 58.5, 90.5,  0.0 },
   { 92.0, 66.0, 91.0,  0.0 },
   { 97.0, 70.5, 66.5,  0.0 },
   { 89.0, 89.5, 81.0,  0.0 },
   {  0.0,  0.0,  0.0,  0.0 }
};
```

Our standard output library.

public class StdOut	
void print(String s)	*print* `s` *to standard output*
void println(String s)	*print* `s` *and a newline to standard output*
void println()	*print a newline to standard output*
void printf(String format, ...)	*print the arguments to standard output, as specified by the format string* `format`

The full StdOut API.



type	code	typical literal	sample format strings	converted string values for output
int	d	512	"%14d"	" 512"
			"%-14d"	"512 "
double	f	1595.1680010754388	"%14.2f"	" 1595.17"
			"%.7f"	"1595.1680011"
	e		"%14.4e"	" 1.5952e+03"
String	s	"Hello, World"	"%14s"	" Hello, World"
			"%-14s"	"Hello, World "
			"%-14.5s"	"Hello "
boolean	b	true	"%b"	"true"

Our standard input library.

public class StdIn

methods for reading individual tokens from standard input

boolean	isEmpty()	*is standard input empty (or only whitespace)?*
int	readInt()	*read a token, convert it to an `int`, and return it*
double	readDouble()	*read a token, convert it to a `double`, and return it*
boolean	readBoolean()	*read a token, convert it to a `boolean`, and return it*
String	readString()	*read a token and return it as a `String`*

methods for reading characters from standard input

boolean	hasNextChar()	*does standard input have any remaining characters?*
char	readChar()	*read a character from standard input and return it*

methods for reading lines from standard input

boolean	hasNextLine()	*does standard input have a next line?*
String	readLine()	*read the rest of the line and return it as a `String`*

methods for reading the rest of standard input

int[]	readAllInts()	*read all remaining tokens and return them as an `int` array*
double[]	readAllDoubles()	*read all remaining tokens and return them as a `double` array*
boolean[]	readAllBooleans()	*read all remaining tokens and return them as a `boolean` array*
String[]	readAllStrings()	*read all remaining tokens and return them as a `String` array*
String[]	readAllLines()	*read all remaining lines and return them as a `String` array*
String	readAll()	*read the rest of the input and return it as a `String`*

The full StdIn API.

Our standard drawing library.

public class StdDraw

drawing commands

```
void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
void square(double x, double y, double radius)
void filledSquare(double x, double y, double radius)
void rectangle(double x, double y, double r1, double r2)
void filledRectangle(double x, double y, double r1, double r2)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)
void text(double x, double y, String s)
```

control commands

`void setXscale(double x0, double x1)`	*reset x-scale to (`x0`, `x1`)*
`void setYscale(double y0, double y1)`	*reset y-scale to (`y0`, `y1`)*
`void setPenRadius(double radius)`	*set pen radius to `radius`*
`void setPenColor(Color color)`	*set pen color to `color`*
`void setFont(Font font)`	*set text font to `font`*
`void setCanvasSize(int w, int h)`	*set canvas size to `w`-by-`h`*
`void enableDoubleBuffering()`	*enable double buffering*
`void disableDoubleBuffering()`	*disable double buffering*
`void show()`	*copy the offscreen canvas to the onscreen canvas*
`void clear(Color color)`	*clear the canvas to color `color`*
`void pause(int dt)`	*pause `dt` milliseconds*
`void save(String filename)`	*save to a `.jpg` or `.png` file*

The full StdDraw API.

Our standard audio library.

```
public class StdAudio
```

`void play(String filename)`	*play the given .wav file*
`void play(double[] a)`	*play the given sound wave*
`void play(double x)`	*play sample for 1/44100 second*
`void save(String filename, double[] a)`	*save to a .wav file*
`double[] read(String filename)`	*read from a .wav file*

The full StdAudio API.

Command line.

```
public class AddInts
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      int sum = 0;
      for (int i = 0; i < n; i++)
      {
         int value = StdIn.readInt();
         sum += value;
      }
      StdOut.println("Sum is " + sum);
   }
}
```

parse command-line argument

read from standard input stream

print to standard output stream

command line

command-line argument

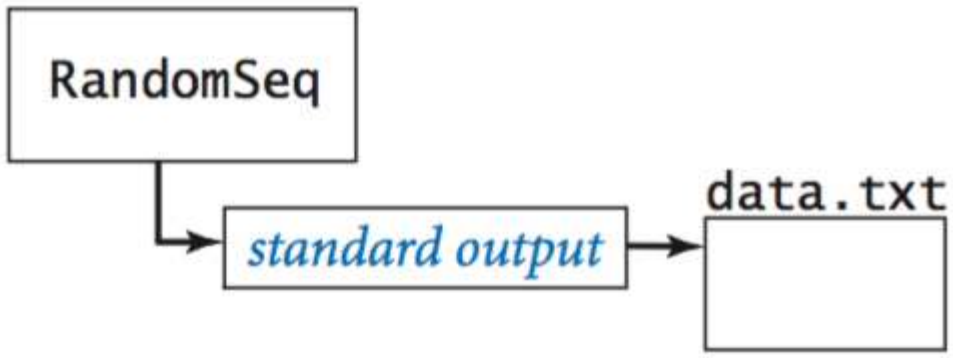
```
% java AddInts 4
144
233
377
1024
Sum is 1778
```

standard input stream

standard output stream

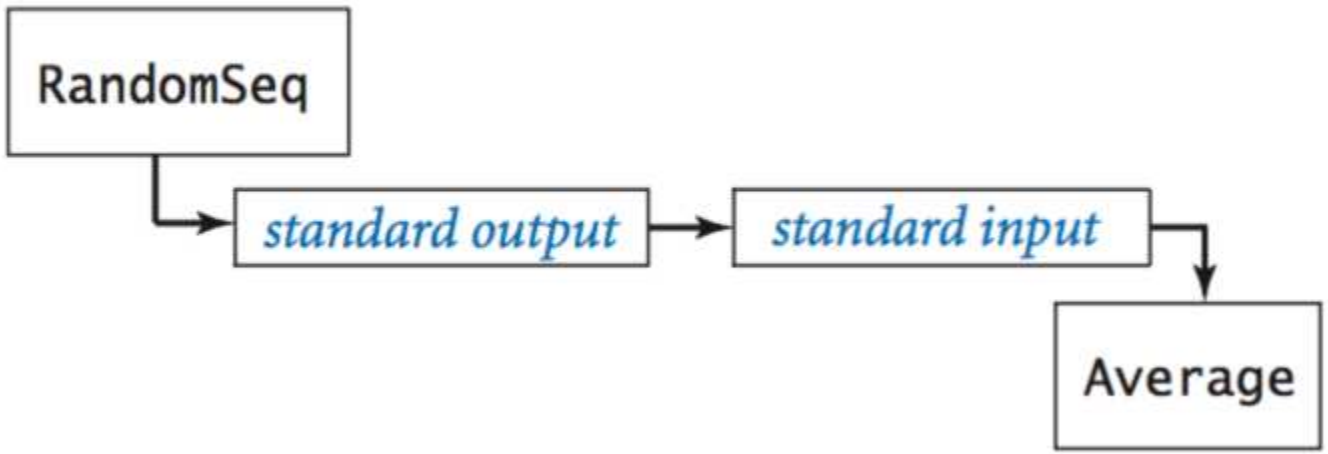
Redirection and piping.

```
% java RandomSeq 1000 > data.txt
```

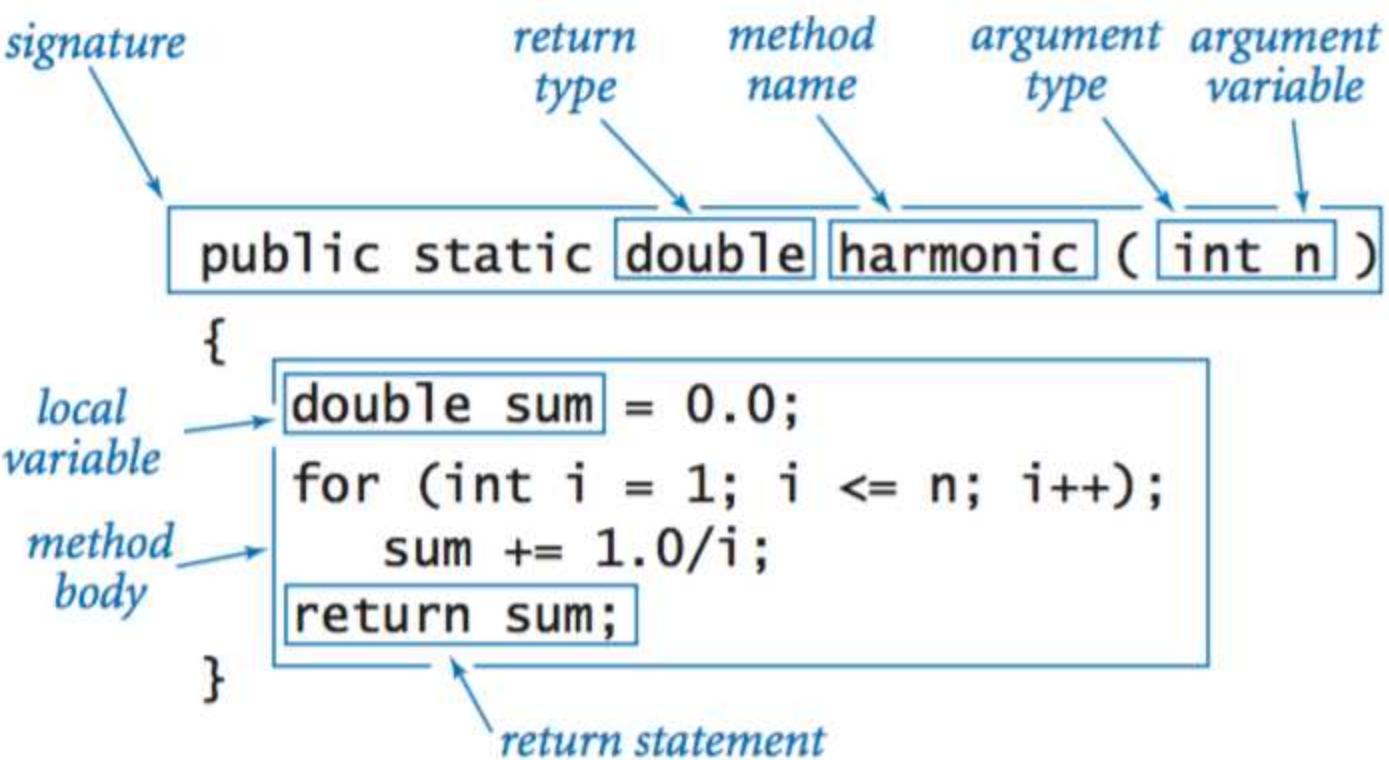



```
% java Average < data.txt
```

```
% java RandomSeq 1000 | java Average
```



Functions.



absolute value of an `int` value

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}
```

absolute value of a `double` value

```
public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

primality test

```
public static boolean isPrime(int n)
{
   if (n < 2) return false;
   for (int i = 2; i <= n/i; i++)
      if (n % i == 0) return false;
   return true;
}
```

hypotenuse of a right triangle

```
public static double hypotenuse(double a, double b)
{  return Math.sqrt(a*a + b*b);  }
```

harmonic number

```
public static double harmonic(int n)
{
   double sum = 0.0;
   for (int i = 1; i <= n; i++)
      sum += 1.0 / i;
   return sum;
}
```

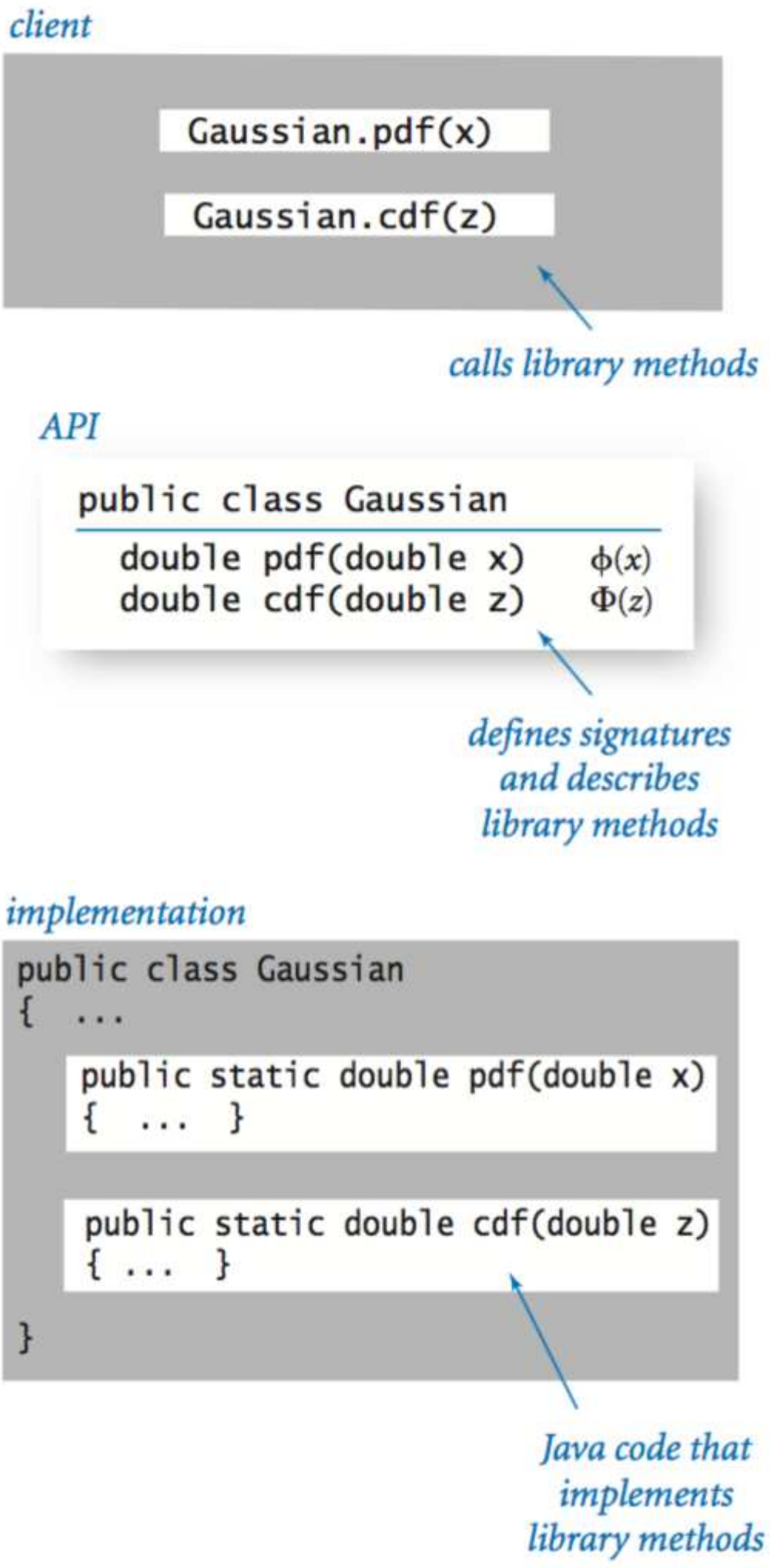
uniform random integer in [0, n)

```
public static int uniform(int n)
{  return (int) (Math.random() * n);  }
```

draw a triangle

```
public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2 )
{
   StdDraw.line(x0, y0, x1, y1);
   StdDraw.line(x1, y1, x2, y2);
   StdDraw.line(x2, y2, x0, y0);
}
```

Libraries of functions.



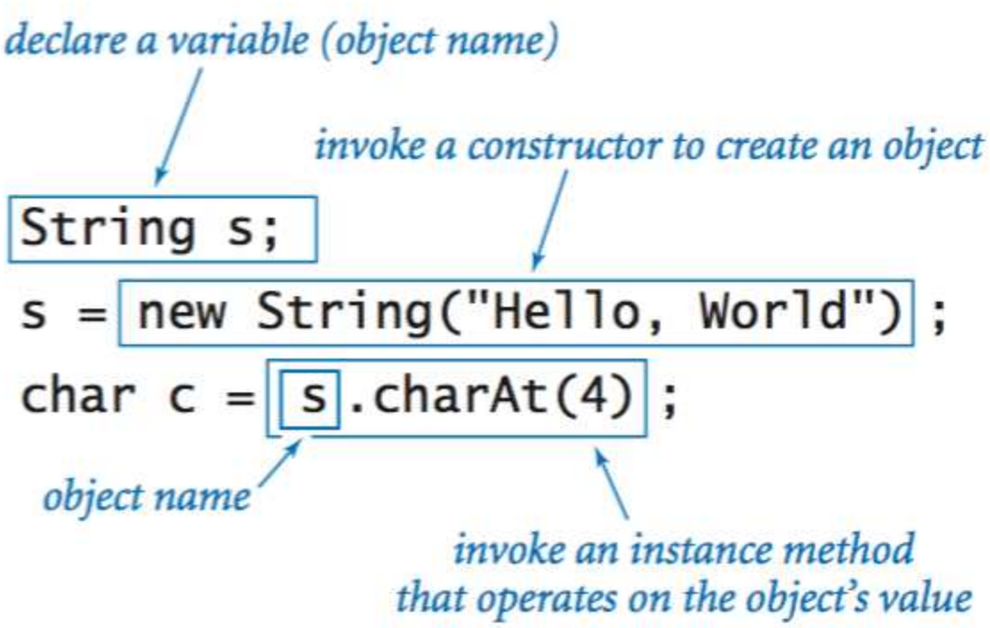
Our standard random library.

public class StdRandom	
void setSeed(long seed)	*set the seed for reproducible results*
int uniform(int n)	*integer between 0 and n-1*
double uniform(double lo, double hi)	*real between lo and hi*
boolean bernoulli(double p)	*true with probability p*
double gaussian()	*normal, mean 0, standard deviation 1*
double gaussian(double mu, double sigma)	*normal, mean mu, standard deviation sigma*
int discrete(double[] probabilities)	*i with probability probabilities[i]*
void shuffle(double[] a)	*randomly shuffle the array a[]*

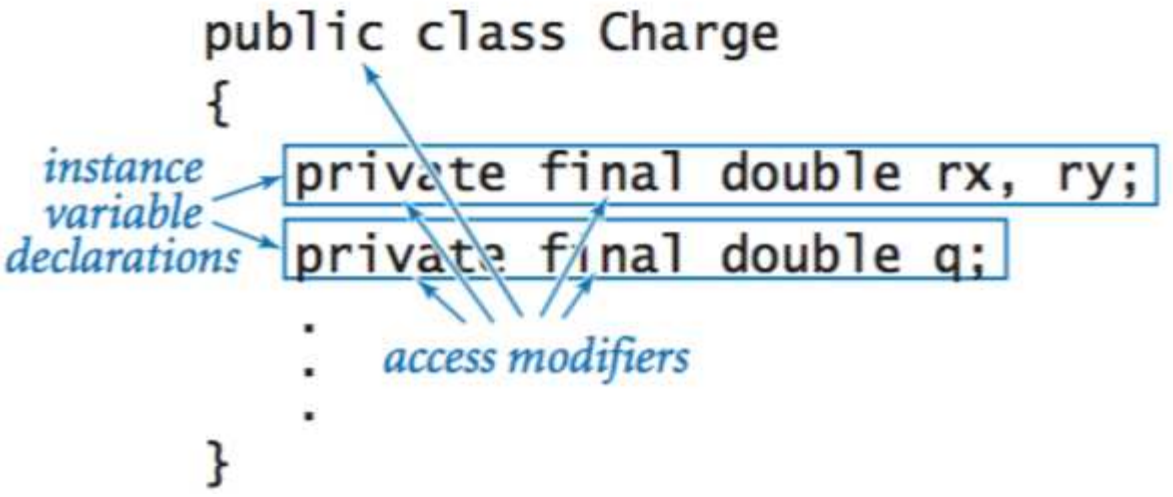
Our standard statistics library.

public class StdStats	
double max(double[] a)	*largest value*
double min(double[] a)	*smallest value*
double mean(double[] a)	*average*
double var(double[] a)	*sample variance*
double stddev(double[] a)	*sample standard deviation*
double median(double[] a)	*median*
void plotPoints(double[] a)	*plot points at (i, a[i])*
void plotLines(double[] a)	*plot lines connecting (i, a[i])*
void plotBars(double[] a)	*plot bars to points at (i, a[i])*

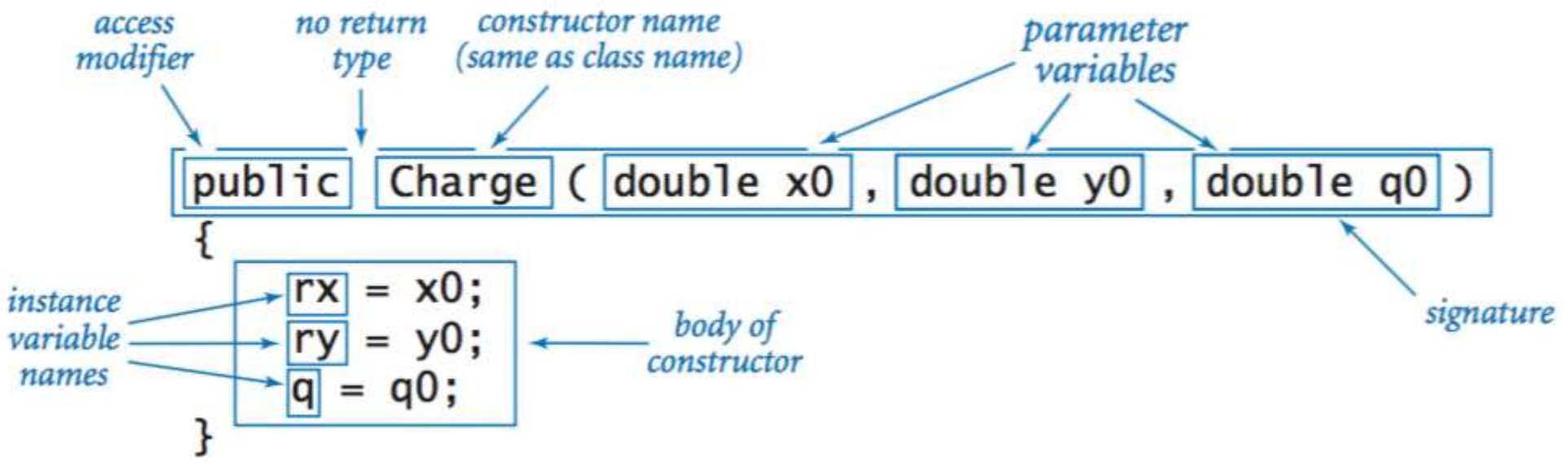
Using an object.



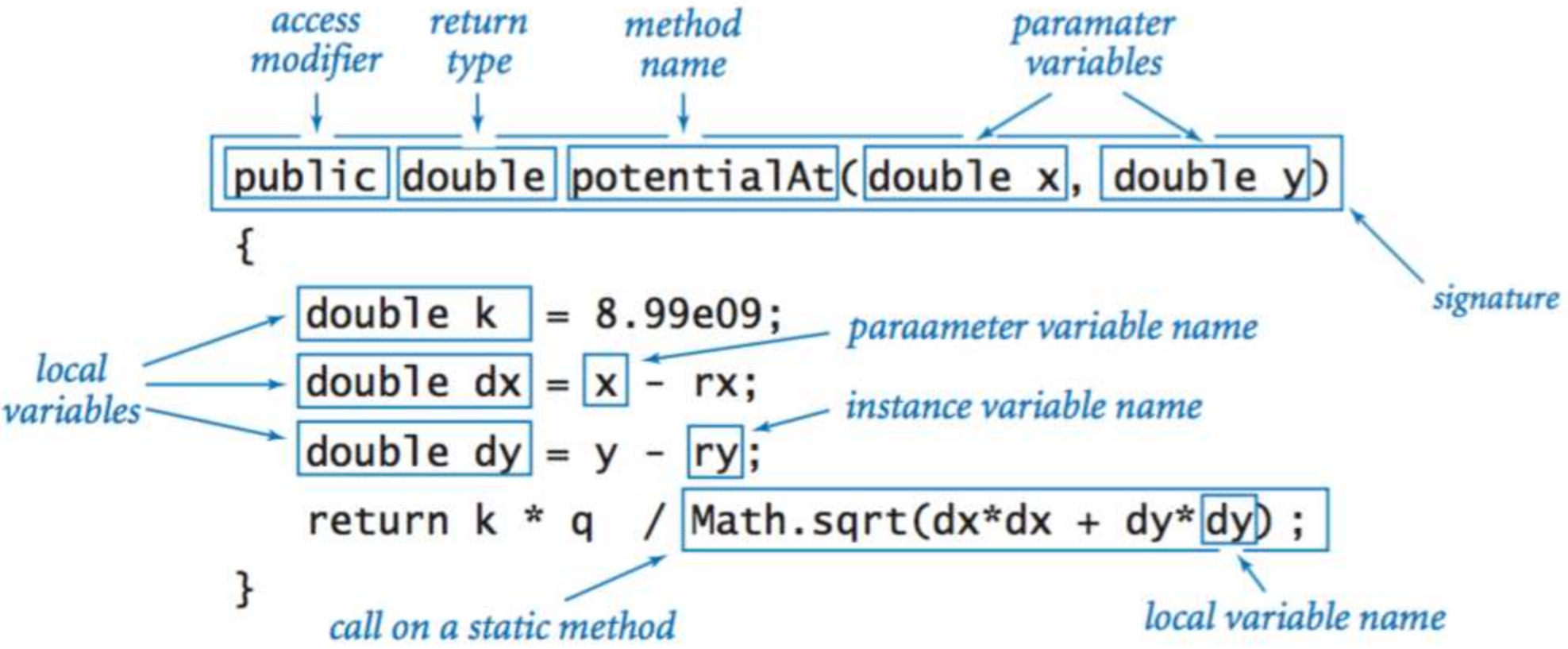
Instance variables.



Constructors.



Instance methods.



Classes.

```
public class Charge
{
   private final double rx, ry;
   private final double q;

   public Charge(double x0, double y0, double q0)
   {  rx = x0; ry = y0; q = q0;  }

   public double potentialAt(double x, double y)
   {
      double k = 8.99e09;
      double dx = x - rx;
      double dy = y - ry;
      return k * q / Math.sqrt(dx*dx + dy*dy);
   }

   public String toString()
   {  return q +" at " + "("+ rx + ", " + ry +")";  }

   public static void main(String[] args)
   {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      Charge c1 = new Charge(0.51, 0.63, 21.3);
      Charge c2 = new Charge(0.13, 0.94, 81.9);
      double v1 = c1.potentialAt(x, y);
      double v2 = c2.potentialAt(x, y);
      StdOut.printf("%.2e\n", (v1 + v2));
   }
}
```

instance variables

class name

constructor

instance methods

instance variable names

test client

create and initialize object

invoke constructor

object name

invoke method

Object-oriented libraries.

client

```
Charge c1 = new Charge(0.51, 0.63, 21.3);

c1.potentialAt(x, y)
```

creates objects and invokes methods

API

```
public class Charge
-----------------------------------------------------------
        Charge(double x0, double y0, double q0)

double potentialAt(double x, double y)   potential at (x, y) due to charge

String toString()                        string representation
```

defines signatures and describes methods

implementation

```
public class Charge
{
   private final double rx, ry;
   private final double q;

   public Charge(double x0, double y0, double q0)
   {  ...  }

   public double potentialAt(double x, double y)
   {  ...  }

   public String toString()
   {  ...  }
}
```

defines instance variables and implements methods

Java's String data type.

public class String

	`String(String s)`	*create a string with the same value as* `s`
	`String(char[] a)`	*create a string that represents the same sequence of characters as in* `a[]`
`int`	`length()`	*number of characters*
`char`	`charAt(int i)`	*the character at index* `i`
`String`	`substring(int i, int j)`	*characters at indices* `i` *through* `(j-1)`
`boolean`	`contains(String substring)`	*does this string contain* `substring`*?*
`boolean`	`startsWith(String prefix)`	*does this string start with* `prefix`*?*
`boolean`	`endsWith(String postfix)`	*does this string end with* `postfix`*?*
`int`	`indexOf(String pattern)`	*index of first occurrence of* `pattern`
`int`	`indexOf(String pattern, int i)`	*index of first occurrence of* `pattern` *after* `i`
`String`	`concat(String t)`	*this string, with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`toLowerCase()`	*this string, with lowercase letters*
`String`	`toUpperCase()`	*this string, with uppercase letters*
`String`	`replace(String a, String b)`	*this string, with* `a`*s replaced by* `b`*s*
`String`	`trim()`	*this string, with leading and trailing whitespace removed*
`boolean`	`matches(String regexp)`	*is this string matched by the regular expression?*
`String[]`	`split(String delimiter)`	*strings between occurrences of* `delimiter`
`boolean`	`equals(Object t)`	*is this string's value the same as* `t`*'s?*
`int`	`hashCode()`	*an integer hash code*

The full java.lang.String API.

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

instance method call	*return type*	*return value*
`a.length()`	`int`	`6`
`a.charAt(4)`	`char`	`'i'`
`a.substring(2, 5)`	`String`	`"w i"`
`b.startsWith("the")`	`boolean`	`true`
`a.indexOf("is")`	`int`	`4`
`a.concat(c)`	`String`	`"now is the"`
`b.replace("t", "T")`	`String`	`"The Time"`
`a.split(" ")`	`String[]`	`{ "now", "is" }`
`b.equals(c)`	`boolean`	`false`

Java's Color data type.

`public class java.awt.Color`

`Color(int r, int g, int b)`	
`int getRed()`	*red intensity*
`int getGreen()`	*green intensity*
`int getBlue()`	*blue intensity*
`Color brighter()`	*brighter version of this color*
`Color darker()`	*darker version of this color*
`String toString()`	*string representation of this color*
`boolean equals(Object c)`	*is this color's value the same as `c`?*

The full java.awt.Color API.

Our input library.

public class In	
In()	*create an input stream from standard input*
In(String name)	*create an input stream from a file or website*
instance methods that read individual tokens from the input stream	
boolean isEmpty()	*is standard input empty (or only whitespace)?*
int readInt()	*read a token, convert it to an `int`, and return it*
double readDouble()	*read a token, convert it to a `double`, and return it*
...	
instance methods that read characters from the input stream	
boolean hasNextChar()	*does standard input have any remaining characters?*
char readChar()	*read a character from standard input and return it*
instance methods that read lines from the input stream	
boolean hasNextLine()	*does standard input have a next line?*
String readLine()	*read the rest of the line and return it as a `String`*
instance methods that read the rest of the input stream	
int[] readAllInts()	*read all remaining tokens; return as array of integers*
double[] readAllDoubles()	*read all remaining tokens; return as array of doubles*
...	

The full In API.

Our output library.

public class Out	
Out()	*create an output stream to standard output*
Out(String name)	*create an output stream to a file*
void print(String s)	*print `s` to the output stream*
void println(String s)	*print `s` and a newline to the output stream*
void println()	*print a newline to the output stream*
void printf(String format, ...)	*print the arguments to the output stream, as specified by the format string `format`*

The full Out API.

Our picture library.

public class Picture

	Picture(String filename)	*create a picture from a file*
	Picture(int w, int h)	*create a blank w-by-h picture*
int	width()	*return the width of the picture*
int	height()	*return the height of the picture*
Color	get(int col, int row)	*return the color of pixel (col, row)*
void	set(int col, int row, Color color)	*set the color of pixel (col, row) to color*
void	show()	*display the picture in a window*
void	save(String filename)	*save the picture to a file*

The full Picture API.

Our stack data type.

public class Stack<Item> implements Iterable<Item>

	Stack()	*create an empty stack*
boolean	isEmpty()	*is the stack empty?*
void	push(Item item)	*push an item onto the stack*
Item	pop()	*return and remove the item that was inserted most recently*
int	size()	*number of items on stack*

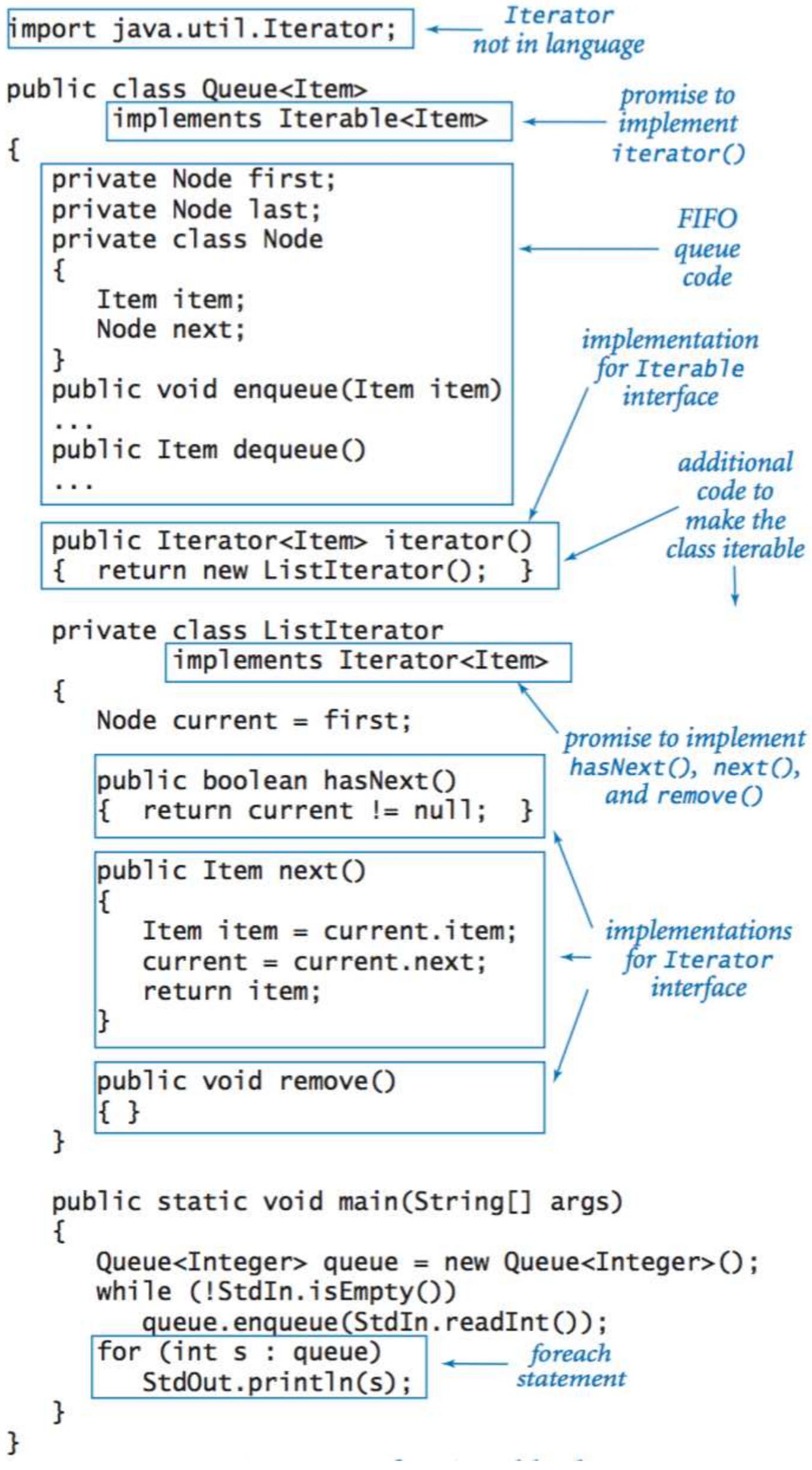
The full Stack API.

Our queue data type.

public class Queue<Item> implements Iterable<Item>	
Queue()	*create an empty queue*
boolean isEmpty()	*is the queue empty?*
void enqueue(Item item)	*insert an item onto queue*
Item dequeue()	*return and remove the item that was inserted least recently*
int size()	*number of items on queue*

The full Queue API.

Iterable.



Our symbol table data type.

public class ST<Key extends Comparable<Key>, Value>	
ST()	*create an empty symbol table*
void put(Key key, Value val)	*associate* `val` *with* `key`
Value get(Key key)	*value associated with* `key`
void remove(Key key)	*remove* `key` *(and its associated value)*
boolean contains(Key key)	*is there a value associated with* `key`*?*
int size()	*number of key–value pairs*
Iterable<Key> keys()	*all keys in the symbol table*

The full ST API.

Our set data type.

public class SET<Key extends Comparable<Key>> implements Iterable<Key>	
SET()	*create an empty set*
boolean isEmpty()	*is the set empty?*
void add(Key key)	*add* `key` *to the set*
void remove(Key key)	*remove* `key` *from set*
boolean contains(Key key)	*is* `key` *in the set?*
int size()	*number of elements in set*

The full SET API.

Our graph data type.

public class Graph	
Graph()	*create an empty graph*
Graph(String filename, String delimiter)	*create graph from a file*
void addEdge(String v, String w)	*add edge v-w*
int V()	*number of vertices*
int E()	*number of edges*
Iterable<String> vertices()	*vertices in the graph*
Iterable<String> adjacentTo(String v)	*neighbors of v*
int degree(String v)	*number of neighbors of v*
boolean hasVertex(String v)	*is v a vertex in the graph?*
boolean hasEdge(String v, String w)	*is v-w an edge in the graph?*

The full Graph API.

Compile-time and run-time errors.

Here's a list of errors compiled by Mordechai Ben-Ari. It includes a list of common error message and typical mistakes that give rise to them.

Last modified on October 30, 2019.

Copyright © 2000–2019 Robert Sedgewick and Kevin Wayne. All rights reserved.

CONTENTS

Core Java

By Cay S. Horstmann; Revised & Updated by Ivan St. Ivanov

ABOUT CORE JAVA

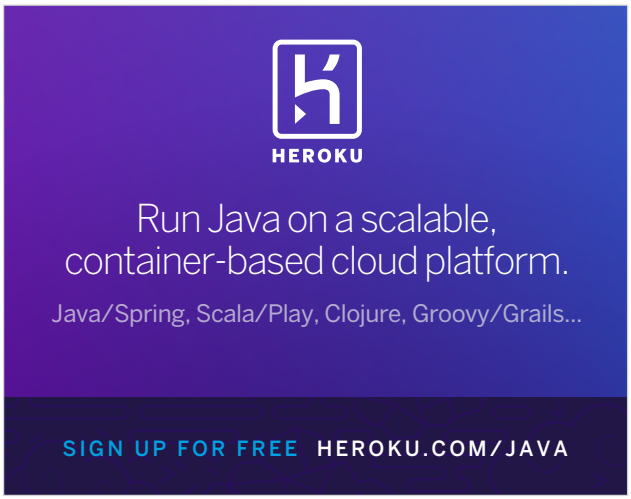
This Refcard gives you an overview of key aspects of the Java language and cheat sheets on the core library (formatted output, collections, regular expressions, logging, properties) as well as the most commonly used tools (`javac`, `java`, `jar`).

JAVA KEYWORDS

KEYWORD	DESCRIPTION	EXAMPLE
abstract	an abstract class or method	`abstract class Writable { public abstract void write(Writer out); public void save(String filename) { ... } }`
assert	with assertions enabled, throws an error if condition not fulfilled	`assert param != null;` **Note:** Run with **-ea** to enable assertions
boolean	the Boolean type with values true and false	`boolean more = false;`
break	breaks out of a switch or loop	`while ((ch = in.next()) != -1) { if (ch == '\n') break; process(ch); }` **Note:** Also see **switch**

byte	the 8-bit integer type	`byte b = -1; // Not the same as 0xFF`
		Note: Be careful with bytes < 0

case	a case of a switch	see **switch**
catch	the clause of a try block catching an exception	see **try**
char	the Unicode character type	`char input = 'Q';`
class	defines a class type	`class Person { private String name; public Person(String aName) { name = aName; } public void print() { System.out.println(name); } }`
continue	continues at the end of a loop	`while ((ch = in.next()) != -1) { if (ch == ' ') continue; process(ch); }`
default	1) the default clause of a switch	see **switch**
default *(cont.)*	2) denotes default implementation of an interface method	`public interface Collection<E> { @Override default Spliterator<E> spliterator() { return Spliterators.spliterator(this, 0); } }`
do	the top of a do/while loop	`do { ch = in.next(); } while (ch == ' ');`
double	the double-precision floating-number type	`double oneHalf = 0.5;`
else	the else clause of an if statement	see **if**
enum	an enumerated type	`enum Mood { SAD, HAPPY };`
extends	defines the parent class of a class	`class Student extends Person { private int id; public Student(String name, int anId) { ... } public void print() { ... } }`
final	a constant, or a class or method that cannot be overridden	`public static final int DEFAULT_ID = 0;`
finally	the part of a try block that is always executed	see **try**
float	the single-precision floating-point type	`float oneHalf = 0.5F;`



KEYWORD	DESCRIPTION	EXAMPLE

for	a loop type	`for (int i = 10; i >= 0; i--)`
		`  System.out.println(i);`
		`for (String s : line.split("\\s+"))`
		`  System.out.println(s);`
		Note: In the "generalized" for loop, the expression after the : must be an array or an `Iterable`

if	a conditional statement	`if (input == 'Q')`
		`  System.exit(0);`
		`else`
		`  more = true;`

implements	defines the interface(s) that a class implements	`class Student`
		`implements Printable {`
		`...`
		`}`

import	imports a package	`import java.util.ArrayList;` `import com.dzone.refcardz.*;`

instanceof	tests if an object is an instance of a class	`if (fred instanceof Student)`
		`  value = ((Student) fred).getId();`
		Note: `null instanceof T` is always false

int	the 32-bit integer type	`int value = 0;`

interface	an abstract type with methods that a class can implement	`interface Printable {`
		`  void print();`
		`}`

long	the 64-bit long integer type	`long worldPopulation = 6710044745L;`
native	a method implemented by the host system	
new	allocates a new object or array	`Person fred = new Person("Fred");`
null	a null reference	`Person optional = null;`
package	a package of classes	`package com.dzone.refcardz;`
private	a feature that is accessible only by methods of this class	see **class**

protected	a feature that is accessible only by methods of this class, its children, and other classes in the same package	`class Student {`
		`  protected int id;`
		`  ...`
		`}`

public	a feature that is accessible by methods of all classes	see **class**
return	returns from a method	`int getId() { return id; }`
short	the 16-bit integer type	`short skirtLength = 24;`

static	a feature that is unique to its class, not to objects of its class	`public class WriteUtil {`
		`  public static void write(Writable[] ws,`
		`  String filename);`
		`  public static final String DEFAULT_EXT = ".dat";`
		`}`

KEYWORD	DESCRIPTION	EXAMPLE
strictfp	Use strict rules for floating-point computations	

super	invoke a superclass constructor or method	`public Student(String name, int anId) {`
		`  super(name); id = anId;`
		`}`
		`public void print() {`
		`  super.print();`
		`  System.out.println(id);`
		`}`

switch	a selection statement	`switch (ch) {`
		`  case 'Q':`
		`  case 'q':`
		`  more = false; break;`
		`  case ' ';`
		`  break;`
		`  default:`
		`  process(ch); break;`
		`}`
		Note: If you omit a break, processing continues with the next case.

synchronized	a method or code block that is atomic to a thread	`public synchronized void addGrade(String gr) {`
		`  grades.add(gr);`
		`}`

this	the implicit argument of a method, or a constructor of this class	`public Student(String id) {this.id = id;}`
		`public Student() {`
		`this(""); }`

throw	throws an exception	`if (param == null)` `  throw new IllegalArgumentException();`
throws	the exceptions that a method can throw	`public void print()` `  throws PrinterException, IOException`

transient	marks data that should not be persistent	`class Student {`
		`  private transient Data cachedData;`
		`  ...`
		`}`

try	a block of code that traps exceptions	`try {`
		`  try {`
		`  fred.print(out);`
		`  } catch (PrinterException ex) {`
		`  ex.printStackTrace();`
		`  }`
		`} finally {`
		`  out.close();`
		`}`

void	denotes a method that returns no value	`public void print() { ... }`

volatile	ensures that a field is coherently accessed by multiple threads	`class Student {`
		`  private volatile int nextId;`
		`  ...`
		`}`

while	a loop	`while (in.hasNext())` `  process(in.next());`

STANDARD JAVA PACKAGES

java.applet	Applets (Java programs that run inside a web page)
java.awt	Graphics and graphical user interfaces

java.beans	Support for JavaBeans components (classes with properties and event listeners)
java.io	Input and output
java.lang	Language support
java.math	Arbitrary-precision numbers
java.net	Networking
java.nio	"New" (memory-mapped) I/O
java.rmi	Remote method invocations
java.security	Security support
java.sql	Database support
java.text	Internationalized formatting of text and numbers
java.time	Dates, time, duration, time zones, etc.
java.util	Utilities (including data structures, concurrency, regular expressions, and logging)

OPERATOR PRECEDENCE

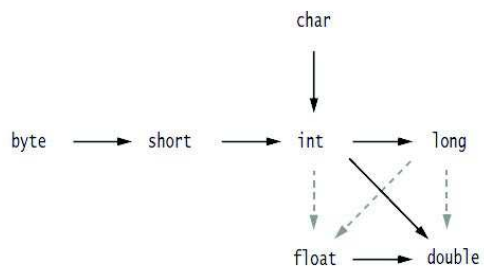
OPERATORS WITH THE SAME PRECEDENCE		NOTES
`[] . ()` (method call)	Left to right	
`! ~ ++ -- +` (unary) `-` (unary) `()` (cast) `new`	Right to left	~ flips each bit of a number
`* / %`	Left to right	Be careful when using % with negative numbers. `-a % b == -(a % b)`, but `a % -b == a % b`. For example, `-7 % 4 == -3`, `7 % -4 == 3`
`+ -`	Left to right	
`<< >> >>>`	Left to right	>> is arithmetic shift (n >> 1 == n / 2 for positive and negative numbers), >>> is logical shift (adding 0 to the highest bits). The right hand side is reduced modulo 32 if the left hand side is an int or modulo 64 if the left hand side is a long. For example, 1 << 35 == 1 << 3
`< <= > >= instanceof`	Left to right	`null instanceof T` is always `false`
`== !=`	Left to right	Checks for identity. Use `equals` to check for structural equality
`&`	Left to right	Bitwise AND; no lazy evaluation with `bool` arguments
`^`	Left to right	Bitwise XOR
`|`	Left to right	Bitwise OR; no lazy evaluation with `bool` arguments
`&&`	Left to right	
`||`	Left to right	
`?:`	Right to left	
`= += -= *= /= %= &= |= ^= <<= >>= >>>=`	Right to left	

PRIMITIVE TYPES

TYPE	SIZE	RANGE	NOTES
`int`	4 bytes	–2,147,483,648 to 2,147,483,647 (just over 2 billion)	The wrapper type is `Integer`. Use `BigInteger` for arbitrary precision integers
`short`	2 bytes	–32,768 to 32,767	
`long`	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Literals end with `L` (e.g. `1L`)
`byte`	1 byte	–128 to 127	Note that the range is not 0 to 255
`float`	4 bytes	approximately ±3.40282347E+38F (6–7 significant decimal digits)	Literals end with `F` (e.g. `0.5F`)
`double`	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)	Use `BigDecimal` for arbitrary precision floating-point numbers
`char`	2 bytes	`\u0000` to `\uFFFF`	The wrapper type is `Character`. Unicode characters > U+FFFF require two `char` values
`boolean`		`true` or `false`	

LEGAL CONVERSIONS BETWEEN PRIMITIVE TYPES

Dotted arrows denote conversions that may lose precision.



LAMBDA EXPRESSIONS

FUNCTIONAL INTERFACES

Interfaces with a single abstract method. Example:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Implementations of this interface can be supplied in-line as a *lambda expression*:

- Anonymous implementations of functional interfaces
- Parameters and body are separated by an arrow sign ("->")
- Parameters of the abstract method are on the left of the arrow
- The implementation is on the right of the arrow

Typical usage of lambda expressions:

```
JButton button = new JButton("MyButton");
button.addActionListener(event ->
doSomeImportantStuff(event));
```

METHOD REFERENCES

Lambda expressions represent anonymous functions. You can pass them as method parameters or return them. The same can be done with named methods using method references.

Typical usage of method references:

WITHOUT METHOD REFERENCE	WITH METHOD REFERENCE
`button.addActionListener(event -> doSomeImportantStuff(event));`	`button.addActionListener(this::doSomeImportantStuff);`
`list.forEach(element -> System.out.println(element));`	`list.forEach(System.out::println);`

There are four kinds of method references:

KIND OF METHOD REFERENCE	EXAMPLE
To a static method	`Collections::emptyList`
To an instance method of a particular (named) object	`user::getFirstName`
To an instance method of an arbitrary object (to be named later) of a given type	`User::getFirstName`
To a constructor	`User::new`

COLLECTIONS & COMMON ALGORITHMS

`ArrayList`	An indexed sequence that grows and shrinks dynamically
`LinkedList`	An ordered sequence that allows efficient insertions and removal at any location
`ArrayDeque`	A double-ended queue that is implemented as a circular array
`HashSet`	An unordered collection that rejects duplicates
`TreeSet`	A sorted set
`EnumSet`	A set of enumerated type values
`LinkedHashSet`	A set that remembers the order in which elements were inserted
`PriorityQueue`	A collection that allows efficient removal of the smallest element
`HashMap`	A data structure that stores key/value associations
`TreeMap`	A map in which the keys are sorted
`EnumMap`	A map in which the keys belong to an enumerated type
`LinkedHashMap`	A map that remembers the order in which entries were added
`WeakHashMap`	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere
`IdentityHashMap`	A map with keys that are compared by ==, not equals

COMMON TASKS

`List<String> strs = new ArrayList<>();`	Collect strings
`strs.add("Hello"); strs.add("World!");`	Add strings
`for (String str : strs) System.out.println(str);`	Do something with all elements in the collection

`Iterator<String> iter = strs.iterator();`
`while (iter.hasNext()) {`
`  String str = iter.next();`
`  if (someCondition(str)) iter.remove();`
`}`
	Remove elements that match a condition. The `remove` method removes the element returned by the preceding call to `next`

`strs.addAll(strColl);`	Add all strings from another collection of strings
`strs.addAll(Arrays.asList(args))`	Add all strings from an array of strings. `Arrays.asList` makes a `List` wrapper for an array
`strs.removeAll(coll);`	Remove all elements of another collection. Uses equals for comparison

`if (0 <= i && i < strs.size()) {`
`  str = strs.get(i);`
`  strs.set(i, "Hello");`
`}`
	Get or set an element at a specified index

`strs.insert(i, "Hello");` `str = strs.remove(i);`	Insert or remove an element at a specified index, shifting the elements with higher index values
`String[] arr = new String[strs.size()];` `strs.toArray(arr);`	Convert from collection to array

`String[] arr = ...;`
`List<String> lst = Arrays.asList(arr);`
`lst = Arrays.asList("foo", "bar", "baz");`
	Convert from array to list. Use the varargs form to make a small collection

`List<String> lst = ...;`
`lst.sort();`
`lst.sort((a, b) ->`
`  a.length() - b.length());`
	Sort a list by the natural order of the elements, or with a custom comparator

`Map<String, Person> map = new LinkedHashMap<String, Person>();`	Make a map that is traversed in insertion order (requires `hashCode` for key type). Use a `TreeMap` to traverse in sort order (requires that key type is comparable)

`for (Map.Entry<String, Person> entry :`
`map.entrySet()) {`
`  String key = entry.getKey();`
`  Person value = entry.getValue();`
`  ...`
`}`
	Iterate through all entries of the map

`Person key = map.get(str);`
`// null if not found`
`map.put(key, value);`
	Get or set a value for a given key

BULK OPERATIONS WITH STREAM API

`strs.forEach(System.out::println);`	Do something with all elements in the collection

`List<String> filteredList = strs`
`  .stream()`
`  .filter(this::someCondition)`
`  .collect(Collectors.toList());`
	Filter elements that match a condition

`String concat = strs .stream() .collect(Collectors.joining(", "));`	Concatenate the elements of a stream
`List<User> users = ...; List<String> firstNames = users .stream() .map(User::getFirstName) .collect(Collectors.toList());`	Create a new list that maps to the original one
`List<String> adminFirstNames = users .stream() .filter(User::isAdmin) .map(User::getFirstName) .collect(Collectors.toList());`	Combine operations This will not result in two traversals of the list elements
`int sumOfAges = users .stream() .mapToLong(User::getAge) .sum();`	Simple reduction operation
`Map<Role, List<User>> byRole = users .stream() .collect(Collectors .groupingBy(User::getRole));`	Group users by a certain attribute
`int sumOfAges = users .parallelStream() .mapToLong(User::getAge) .sum();`	All the above operations can be done in parallel

CHARACTER ESCAPE SEQUENCES

`\b`	backspace `\u0008`
`\t`	tab `\u0009`
`\n`	newline `\u000A`
`\f`	form feed `\u000C`
`\r`	carriage return `\u000D`
`\"`	double quote
`\'`	single quote
`\\`	backslash
`\uhhhh` (`hhhh` is a hex number between `0000` and `FFFF`)	The UTF-16 code point with value `hhhh`
`\ooo` (`ooo` is an octal number between `0` and `377`)	The character with octal value `ooo`

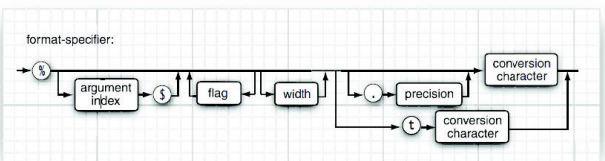
Note: Unlike in C/C++, `\xhh` is not allowed.

FORMATTED OUTPUT WITH PRINTF

TYPICAL USAGE

```
System.out.printf("%4d %8.2f", quantity, price);
String str = String.format("%4d %8.2f", quantity, price);
```

Each format specifier has the following form. See the tables for flags and conversion characters.



FLAGS

FLAG	DESCRIPTION	EXAMPLE
`+`	Prints sign for positive and negative numbers	`+3333.33`
space	Adds a space before positive numbers	`| 3333.33|`
`0`	Adds leading zeroes	`003333.33`
`-`	Left-justifies field	`|3333.33 |`
`(`	Encloses negative number in parentheses	`(3333.33)`
`,`	Adds group separators	`3,333.33`
`#` (for `f` format)	Always includes a decimal point	`3,333.`
`#` (for `x` or `o` format)	Adds 0x or 0 prefix	`0xcafe`
`$`	Specifies the index of the argument to be formatted; for example, `%1$d %1$x` prints the first argument in decimal and hexadecimal	`159 9F`
`<`	Formats the same value as the previous specification; for example, `%d %<x` prints the same number in decimal and hexadecimal	`159 9F`

CONVERSION CHARACTERS

CONVERSION CHARACTER	DESCRIPTION	EXAMPLE
d	Decimal integer	`159`
x	Hexadecimal integer	`9f`
o	Octal integer	`237`
f	Fixed-point floating-point	`15.9`
e	Exponential floating-point	`1.59e+01`
g	General floating-point (the shorter of e and f)	
a	Hexadecimal floating-point	`0x1.fccdp3`
s	String	`Hello`
c	Character	`H`
b	boolean	`true`
h	Hash code	`42628b2`
t*x*	Date and time	See next table
%	The percent symbol	`%`
n	The platform-dependent line separator	

FORMATTED OUTPUT WITH MESSAGEFORMAT

Typical usage:

```
String msg = MessageFormat.format("On {1, date, long}, a
{0} caused {2,number,currency} of damage.", "hurricane",
new GregorianCalendar(2009, 0, 15). getTime(), 1.0E8);
```

Yields `"On January 1, 1999, a hurricane caused $100,000,000 of damage"`

- The nth item is denoted by {n, format, subformat} with optional formats and subformats shown below
- {0} is the first item
- The following table shows the available formats

- Use single quotes for quoting, for example '{' for a literal left curly brace
- Use '' for a literal single quote

FORMAT	SUBFORMAT	EXAMPLE
number	none	1,234.567
	integer	1,235
	currency	$1,234.57
	percent	123,457%
date	none or medium	Jan 15, 2015
	short	1/15/15
	long	January 15, 2015
	full	Thursday, January 15, 2015
time	none or medium	3:45:00 PM
	short	3:45 PM
	long	3:45:00 PM PST
	full	3:45:00 PM PST
choice	List of choices, separated by |. Each choice has • a lower bound (use -\u221E for −∞) • a relational operator: < for "less than", # or \u2264 for ≤ • a message format string For example, {1,choice,0#no houses|1#one house|2#{1} houses}	no houses one house 5 houses

REGULAR EXPRESSIONS

COMMON TASKS

`String[] words = str.split("\\s+");`	Split a string along white space boundaries

`Pattern pattern = Pattern.compile("[0-9]+");`
`Matcher matcher = pattern.matcher(str);`
`String result = matcher.replaceAll("#");`
	Replace all matches. Here we replace all digit sequences with a #.

`Pattern pattern = Pattern.compile("[0-9]+");`
`Matcher matcher = pattern.matcher(str);`
`while (matcher.find()) {`
`  process(str.substring(matcher.start(), matcher.end()));`
`}`
	Find all matches.

`Pattern pattern = Pattern.compile("(1?[0-9]):([0-5][0-9])[ap]m");`
`Matcher matcher = pattern.matcher(str);`
`for (int i = 1; i <= matcher.groupCount(); i++) {`
`  process(matcher.group(i));`
`}`
	Find all groups (indicated by parentheses in the pattern). Here we find the hours and minutes in a date.

REGULAR EXPRESSION SYNTAX

CHARACTERS	
c	The character *c*
\u*nnnn*, \x*nn*, \0*n*, \0*nn*, \0*nnn*	The code unit with the given hex or octal value
\t, \n, \r, \f, \a, \e	The control characters tab, newline, return, form feed, alert, and escape
\c*c*	The control character corresponding to the character *c*
CHARACTER CLASSES	
[C_1C_2 ...]	**Union:** Any of the characters represented by $C_1C_2, \ldots$ The C_i are characters, character ranges c_1-c_2, or character classes. Example: [a-zA-Z0-9_]
[^C_1C_2 ...]	**Complement:** Characters not represented by any of $C_1C_2, \ldots$ Example: [^0-9]
[C_1&&C_2&& ...]	**Intersection:** Characters represented by all of $C_1C_2, \ldots$ Example: [A-f&&[^G-`]]
PREDEFINED CHARACTER CLASSES	
.	Any character except line terminators (or any character if the DOTALL flag is set)
\d	A digit [0-9]
\D	A nondigit [^0-9]
\s	A whitespace character [\t\n\r\f\x0B]
\S	A nonwhitespace character
\w	A word character [a-zA-Z0-9_]
\W	A nonword character
\p{*name*}	A named character class—see table below
\P{*name*}	The complement of a named character class
BOUNDARY MATCHERS	
^ $	Beginning, end of input (or beginning, end of line in multiline mode)
\b	A word boundary
\B	A nonword boundary
\A	Beginning of input
\z	End of input
\Z	End of input except final line terminator
\G	End of previous match
QUANTIFIERS	
X?	Optional *X*
*X**	*X*, 0 or more times
X+	*X*, 1 or more times
X{*n*} *X*{*n*,} *X*{*n*,*m*}	*X* *n* times, at least *n* times, between *n* and *m* times
QUANTIFIER SUFFIXES	
?	Turn default (greedy) match into reluctant match
+	Turn default (greedy) match into reluctant match
SET OPERATIONS	
XY	Any string from *X*, followed by any string from *Y*
X|*Y*	Any string from *X* or *Y*

GROUPING	
(*X*)	Capture the string matching *X* as a group
\g	The match of the *g*th group
ESCAPES	
\c	The character c (must not be an alphabetic character)
\Q ... \E	Quote . . . verbatim
(? ...)	Special construct—see API notes of Pattern class

PREDEFINED CHARACTER CLASS NAMES

Lower	ASCII lower case [a-z]
Upper	ASCII upper case [A-Z]
Alpha	ASCII alphabetic [A-Za-z]
Digit	ASCII digits [0-9]
Alnum	ASCII alphabetic or digit [A-Za-z0-9]
XDigit	Hex digits [0-9A-Fa-f]
Print or Graph	Printable ASCII character [\x21-\x7E]
Punct	ASCII nonalpha or digit [\p{Print}&&\P{Alnum}]
ASCII	All ASCII [\x00-\x7F]
Cntrl	ASCII Control character [\x00-\x1F]
Blank	Space or tab [\t]
Space	Whitespace [\t\n\r\f\0x0B]
javaLowerCase	Lower case, as determined by Character.isLowerCase()
javaUpperCase	Upper case, as determined by Character.isUpperCase()
javaWhitespace	White space, as determined by Character.isWhitespace()
javaMirrored	Mirrored, as determined by Character.isMirrored()
In*Block*	*Block* is the name of a Unicode character block, with spaces removed, such as BasicLatin or Mongolian
Category or In*Category*	*Category* is the name of a Unicode character category such as L (letter) or Sc (currency symbol)

FLAGS FOR MATCHING

The pattern matching can be adjusted with flags, for example:

```
Pattern pattern = Pattern.compile(patternString,
Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE)
```

FLAG	DESCRIPTION
CASE_INSENSITIVE	Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account
UNICODE_CASE	When used in combination with CASE_INSENSITIVE, use Unicode letter case for matching
MULTILINE	^ and $ match the beginning and end of a line, not the entire input
UNIX_LINES	Only '\n' is recognized as a line terminator when matching ^ and $ in multiline mode
DOTALL	When using this flag, the . symbol matches all characters, including line terminators
CANON_EQ	Takes canonical equivalence of Unicode characters into account. For example, u followed by ¨ (diaeresis) matches ü
LITERAL	The input string that specifies the pattern is treated as a sequence of literal characters, without special meanings for . [ ] etc

LOGGING

COMMON TASKS

Logger logger = Logger.getLogger("com.mycompany.myprog.mycategory");	Get a logger for a category
logger.info("Connection successful.");	Logs a message of level FINE. Available levels are SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, with corresponding methods severe, warning, and so on
logger.log(Level.SEVERE, "Unexpected exception", Throwable);	Logs the stack trace of a Throwable
logger.setLevel(Level.FINE);	Sets the logging level to FINE. By default, the logging level is INFO, and less severe logging messages are not logged
Handler handler = new FileHandler("%h/myapp.log", SIZE_LIMIT, LOG_ROTATION_COUNT); handler.setFormatter(new SimpleFormatter()); logger.addHandler(handler);	Adds a file handler for saving the log records in a file. See the table below for the naming pattern. This handler uses a simple formatter instead of the XML formatter that is the default for file handlers

LOGGING CONFIGURATION FILES

The logging configuration can be configured through a logging configuration file, by default jre/lib/logging.properties. Another file can be specified with the system property java.util.logging.config.file when starting the virtual machine. (Note that the LogManager runs before main.)

CONFIGURATION PROPERTY	DESCRIPTION	DEFAULT
loggerName.level	The logging level of the logger by the given name	None; the logger inherits the handler from its parent
handlers	A whitespace or comma-separated list of class names for the root logger. An instance is created for each class name, using the default constructor	java.util.logging.ConsoleHandler
loggerName.handlers	A whitespace or comma-separated list of class names for the given logger	None
loggerName.useParentHandlers	false if the parent logger's handlers (and ultimately the root logger's handlers) should not be used	true
config	A whitespace or comma-separated list of class names for initialization	None

CONFIGURATION PROPERTY	DESCRIPTION	DEFAULT
java.util.logging.FileHandler.level java.util.logging.ConsoleHandler.level	The default handler level	Level.ALL for FileHandler, Level.INFO for ConsoleHandler
java.util.logging.FileHandler.filter java.util.logging.ConsoleHandler.filter	The class name of the default filter	None
java.util.logging.FileHandler.formatter java.util.logging.ConsoleHandler.formatter	The class name of the default formatter	java.util.logging.XMLFormatter for FileHandler, java.util.logging.SimpleFormatter for ConsoleHandler
java.util.logging.FileHandler.encoding java.util.logging.ConsoleHandler.encoding	The default encoding	default platform encoding
java.util.logging.FileHandler.limit	The default limit for rotating log files, in bytes	0 (No limit), but set to 50000 in *jre*/lib/logging.properties
java.util.logging.FileHandler.count	The default number of rotated log files	1
java.util.logging.FileHandler.pattern	The default naming pattern for log files. The following tokens are replaced when the file is created:	%h/java%u.log
java.util.logging.FileHandler.append	The default append mode for file loggers; true to append to an existing log file	false

TOKEN	DESCRIPTION
/	Path seperator
%t	System temporary directory
%h	Value of user.home system property
%g	The generation number of rotated logs
%u	A unique number for resolving naming conflicts
%%	The % character

PROPERTY FILES

- Contain name/value pairs, separated by =, :, or whitespace
- Whitespace around the name or before the start of the value is ignored
- Lines can be continued by placing an \ as the last character; leading whitespace on the continuation line is ignored

```
button1.tooltip = This is a long \
                  tooltip text.
```

- \t \n \f \r \\ \uxxxx escapes are recognized (but not \b or octal escapes)
- Files are assumed to be encoded in ISO 8859-1; use native2ascii to encode non-ASCII characters into Unicode escapes
- Blank lines and lines starting with # or ! are ignored

Typical usage:

```
Properties props = new Properties();
props.load(new FileInputStream("prog.properties"));
String value = props.getProperty("button1.tooltip");
// null if not present
```

Also used for resource bundles:

```
ResourceBundle bundle = ResourceBundle.getBundle("prog");
  // Searches for prog_en_US.properties,
  // prog_en.properties, etc.
String value = bundle.getString("button1.tooltip");
```

JAR FILES

- Used for storing applications, code libraries
- By default, class files and other resources are stored in ZIP file format
- META-INF/MANIFEST.MF contains JAR metadata
- META-INF/services can contain service provider configuration
- Use the jar utility to make JAR files

JAR UTILITY OPTIONS

OPTION	DESCRIPTION
c	Creates a new or empty archive and adds files to it. If any of the specified file names are directories, the jar program processes them recursively
C	Temporarily changes the directory. For example, jar cvfC myprog.jar classes *.class changes to the classes subdirectory to add class files
e	Creates a Main-Class entry in the manifest jar cvfe myprog.jar com.mycom.mypkg.MainClass files
f	Specifies the JAR file name as the second command-line argument. If this parameter is missing, jar will write the result to standard output (when creating a JAR file) or read it from standard input (when extracting or tabulating a JAR file)
i	Creates an index file (for speeding up lookups in a large archive)
m	Adds a manifest to the JAR file jar cvfm myprog.jar mymanifest.mf files
M	Does not create a manifest file for the entries
t	Displays the table of contents jar tvf myprog.jar
u	Updates an existing JAR file jar uf myprog.jar com/mycom/mypkg/SomeClass.class
v	Generates verbose output
x	Extracts files. If you supply one or more file names, only those files are extracted. Otherwise, all files are extracted jar xf myprog.jar
0	Stores without ZIP compression

COMMON JAVAC OPTIONS

OPTION	DESCRIPTION
`-cp` or `-classpath`	Sets the class path, used to search for class files. The class path is a list of directories, JAR files, or expressions of the form directory/'*' (Unix) or directory* (Windows). The latter refers to all JAR files in the given directory. Class path items are separated by : (Unix) or ; (Windows). If no class path is specified, it is set to the current directory. If a class path is specified, the current directory is not automatically included—add a . item if you want to include it
`-sourcepath`	Sets the path used to search for source files. If source and class files are present for a given file, the source is compiled if it is newer. If no source path is specified, it is set to the current directory
`-d`	Sets the path used to place the class files. Use this option to separate `.java` and `.class` files
`-source`	Sets the source level. Valid values are 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 5, 6, 7, 8
`-deprecation`	Gives detail information about the use of deprecated features
`-Xlint:unchecked`	Gives detail information about unchecked type conversion warnings

COMMON JAVA OPTIONS

OPTION	DESCRIPTION
`-cp` or `-classpath`	Sets the class path, used to search for class files. See the previous table for details. Note that `javac` can succeed when `java` fails if the current directory is on the source path but not the class path
`-ea` or `-enableassertions`	Enable assertions. By default, assertions are disabled
`-D`*`property=value`*	Sets a system property that can be retrieved by `System.getProperty(String)`
`-jar`	Runs a program contained in a JAR file whose manifest has a `Main-Class` entry. When this option is used, the class path is ignored
`-verbose`	Shows the classes that are loaded. This option may be useful to debug class loading problems
`-Xms`*`size`* `-Xmx`*`size`*	Sets the initial or maximum heap size. The size is a value in bytes. Add a suffix `k` or `m` for kilobytes or megabytes, for example, `-Xmx10m`

Core Java Concurrency

ORIGINAL BY **ALEX MILLER** UPDATE BY **IGOR SOROKIN**

CONTENTS

INTRODUCTION

From its creation, Java has supported key concurrency concepts such as threads and locks. This Refcard will help Java developers working with multi-threaded programs to understand core concurrency concepts and how to apply them.

CONCEPTS

CONCEPT	DESCRIPTION
Atomicity	An atomic operation is one which is executed in an all or nothing fashion, therefore partial state is impossible.
Visibility	The conditions when one thread sees changes made by another thread

Table 1: *Concurrency concepts*

RACE CONDITION

A race condition occurs when more than one thread is performing a series of actions on shared resources, and several possible outcomes can exist based on the order of the actions from each thread. The below code is not thread-safe and the value could be initialized more than once, as `check-then-act` (check for `null`, then initialize) that lazily initializes the field is not **atomic**.

```
class Lazy <T> {
  private volatile T value;

  T get() {
    if (value == null)
      value = initialize();
    return value;
  }
}
```

DATA RACE

A data race occurs when 2 or more threads try to access the same non-final variable without synchronization. Not using synchronization may lead to making changes which are **not visible** to other threads, so reading the stale data is possible, which in turn may have consequences such as infinite loops, corrupted data structures, or inaccurate computations. This code might result in an infinite loop, because the reader thread may never observe the changes made by the writer threads:

```
class Waiter implements Runnable {
  private boolean shouldFinish;

  void finish() { shouldFinish = true; }

  public void run() {
    long iteration = 0;
    while (!shouldFinish) {
      iteration++;
    }
    System.out.println("Finished after: " +
iteration);
  }
}

class DataRace {

  public static void main(String[] args)
    throws InterruptedException {
    Waiter waiter = new Waiter();
    Thread waiterThread = new Thread(waiter);
    waiterThread.start();

    waiter.finish();
    waiterThread.join();
  }
}
```

JAVA MEMORY MODEL: HAPPENS-BEFORE RELATIONSHIP

The Java memory model is defined in terms of actions like reading and writing fields, and synchronizing on a monitor. Actions can be ordered by a **happens-before relationship**, that can be used to reason about when a thread sees the result of another thread's actions, and what constitutes a properly synchronized program.

Happens-before relationships have the following properties:

- The invocation of `Thread#start` happens before any action in this thread.
- Releasing a monitor happens before any subsequent acquisition of the same monitor.
- A write to a volatile variable happens before any subsequent read of a volatile variable.
- A write to a final variable happens before the reference of the object is published.

- All actions in a thread happen before returning from a `Thread#join` on that thread.

In Image 1, `Action X` happens before `Action Y`, therefore in `Thread 2` all operations to the right of `Action Y` will see all the operations to the left of `Action X` in `Thread 1`.

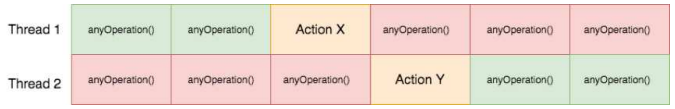


Image 1: *Happens-before illustration*

STANDARD SYNCHRONIZATION FEATURES

THE `SYNCHRONIZED` KEYWORD

The `synchronized` keyword is used to prevent different threads executing the same code block simultaneously. It guarantees that since you acquire a lock (by entering the synchronized block), data, which is protected by this lock, can be manipulated in exclusive mode, so the operation can be **atomic**. Also, it guarantees that other threads will observe the result of the operation after they acquire the same lock.

```
class AtomicOperation {
  private int counter0;
  private int counter1;

  void increment() {
    synchronized (this) {
      counter0++;
      counter1++;
    }
  }
}
```

The synchronized keyword can be also specified on a method level.

TYPE OF METHOD	REFERENCE WHICH IS USED AS A MONITOR
static	The class object of the class with the method
non-static	The `this` reference

Table 2: *Monitors, which are used when the whole method is synchronized*

```
class Reentrantcy {

  synchronized void doAll() {
    doFirst();
    doSecond();
  }

  synchronized void doFirst() {
    System.out.println("First operation is" +
    "successful.");
  }

  synchronized void doSecond() {
    System.out.println("Second operation is" +
    "successful.");
  }
}
```

The level of contention affects how the monitor is acquired:

STATE	DESCRIPTION
init	Just created, never acquired.
biased	There is no contention and the code protected by the lock is executed only by one thread. The cheapest one to acquire.
thin	Monitor is acquired by several threads with no contention. Relatively cheap CAS is used for taking the lock.
fat	There is contention. The JVM requests an OS mutex and lets the OS scheduler handle thread-parking and wake ups.

Table 3: *Monitor states*

WAIT/NOTIFY

`wait/notify/notifyAll` methods are declared in the `Object` class. `wait` is used to make a thread to advance to the `WAITING` or `TIMED_WAITING` (if the time-out value is passed) status. In order to wake up a thread, any of these actions can be done:

- Another thread invokes `notify`, which wakes up an arbitrary thread waiting on the monitor.
- Another thread invokes `notifyAll`, which wakes up all the threads waiting on the monitor.
- `Thread#interrupt` is invoked. In this case, `InterruptedException` is thrown.

The most common pattern is a condition loop:

```
class ConditionLoop {
  private boolean condition;

  synchronized void waitForCondition()
    throws InterruptedException {
    while (!condition) {
      wait();
    }
  }

  synchronized void satisfyCondition() {
    condition = true;
    notifyAll();
  }
}
```

- Keep in mind that in order to use `wait/notify/notifyAll` on an object, you need to acquire the lock on this object first.
- Always wait inside a loop that checks the condition being waited on – this addresses the timing issue if another thread satisfies the condition before the wait begins. Also, it protects your code from spurious wake-ups that can (and do) occur.
- Always ensure that you satisfy the waiting condition before calling `notify/notifyAll`. Failing to do so will cause a notification but no thread will ever be able to escape its wait loop.

THE VOLATILE KEYWORD

`volatile` solves the problem of **visibility**, and makes changes of the variable's value to be **atomic**, because there is a happens-before relationship: write to a volatile variable happens before any subsequent read from the volatile variable. Therefore, it guarantees that any subsequent reads of the field will see the value, which was set by the most recent write.

```
class VolatileFlag implements Runnable {

  private volatile boolean shouldStop;

  public void run() {
    while (!shouldStop) {
      //do smth
    }
    System.out.println("Stopped.");
  }

  void stop() {
    shouldStop = true;
  }

  public static void main(String[] args) throws
InterruptedException {
    VolatileFlag flag = new VolatileFlag();
    Thread thread = new Thread(flag);
    thread.start();

    flag.stop();
    thread.join();
  }
}
```

ATOMICS

The `java.util.concurrent.atomic` package contains a set of classes that support atomic compound actions on a single value in a lock-free manner similar to `volatile`.

Using AtomicXXX classes, it is possible to implement an atomic `check-then-act` operation:

```
class CheckThenAct {
  private final AtomicReference<String> value =
new AtomicReference<>();

  void initialize() {
    if (value.compareAndSet(null, "value")) {
      System.out.println("Initialized only once.");
    }
  }
}
```

Both `AtomicInteger` and `AtomicLong` have atomic `increment/decrement` operation:

```
class Increment {
  private final AtomicInteger state =
new AtomicInteger();

  void advance() {
    int oldState = state.getAndIncrement();
    System.out.println("Advanced: '" + oldState +
 "' -> '" + (oldState + 1) + "'.");
  }
}
```

If you want to have a counter and do not need to get its value atomically, consider using `LongAdder` instead of `AtomicLong/AtomicInteger`. `LongAdder` maintains the value across several cells and grows their number if it's needed, consequently it performs better under high contention.

THREADLOCAL

One way to contain data within a thread and make locking unnecessary is to use ThreadLocal storage. Conceptually, ThreadLocal acts as if there is a variable with its own version in every Thread. ThreadLocals are commonly used for stashing per-Thread values like the "current transaction" or other resources. Also, they are used to maintain per-thread counters, statistics, or ID generators.

```
class TransactionManager {
  private final
  ThreadLocal<Transaction> currentTransaction
  = ThreadLocal.withInitial(NullTransaction::new);

  Transaction currentTransaction() {
    Transaction current = currentTransaction.get();
    if (current.isNull()) {
      current = new TransactionImpl();
      currentTransaction.set(current);
    }
    return current;
  }
}
```

SAFE PUBLICATION

Publishing an object is making its reference available outside of the current scope (for example: return a reference from a getter). Ensuring that object is published safely (only when it is fully constructed) may require synchronization. The safe publication could be achieved using:

- **Static initializers**. Only one thread can initialize static variables because initialization of the class is done under an exclusive lock.

```
class StaticInitializer {
  // Publishing an immutable object without
  //additional initialization
  public static final Year year = Year.of(2017);
  public static final Set<String> keywords;

  // Using static initializer to construct a
  //complex object
  static {
    // Creating mutable set
    Set<String> keywordsSet = new HashSet<>();
    // Initializing state
    keywordsSet.add("java");
    keywordsSet.add("concurrency");
    // Making set unmodifiable
    keywords = Collections.
unmodifiableSet(keywordsSet);
  }
}
```

- **Volatile field**. The reader thread will always read the most

recent value because a write to a volatile variable **happens before** any subsequent read.

```
class Volatile {
  private volatile String state;

  void setState(String state) {
    this.state = state;
  }

  String getState() {
    return state;
  }
}
```

- **Atomics**. For example, `AtomicInteger` stores the value in a volatile field, so the same rule for volatile variables is applicable here.

```
class Atomics {
  private final AtomicInteger state =
new AtomicInteger();

  void initializeState(int state) {
    this.state.compareAndSet(0, state);
  }

  int getState() {
    return state.get();
  }
}
```

- **Final Fields**

```
class Final {
  private final String state;

  Final(String state) {
    this.state = state;
  }

  String getState() {
    return state;
  }
}
```

Make sure that the this reference is not escaped during construction.

```
class ThisEscapes {
 private final String name;

 ThisEscapes(String name) {
   Cache.putIntoCache(this);
   this.name = name;
 }

 String getName() { return name; }
}

class Cache {
 private static final Map<String, ThisEscapes>
CACHE = new ConcurrentHashMap<>();

 static void putIntoCache(
ThisEscapes thisEscapes) {
   // 'this' reference escaped before the object
//is fully constructed.

 CACHE.putIfAbsent(thisEscapes.getName(),
thisEscapes);
 }
}
```

- Correctly synchronized field.

```
class Synchronization {

  private String state;

  synchronized String getState() {
    if (state == null)
      state = "Initial";
    return state;
  }
}
```

IMMUTABLE OBJECTS

A great property of immutable objects is that they are thread-safe, so no synchronization is necessary. The requirements for an object to be immutable are:

- All fields are final.
- All fields must be either mutable or immutable objects too, but do not escape the scope of the object so the state of the object cannot be altered after construction.
- this reference does not escape during construction.
- The class is final, so it is not possible to override this behavior in subclasses.

Example of an immutable object:

```
// Marked as final - subclassing is forbidden
public final class Artist {
  // Immutable object, field is final
  private final String name;
  // Collection of immutable objects, field is final
  private final List<Track> tracks;

 public Artist(String name, List<Track> tracks) {
    this.name = name;
     // Defensive copy
    List<Track> copy = new ArrayList<>(tracks);
    // Making mutable collection unmodifiable
    this.tracks = Collections.unmodifiableList(copy);
    // 'this' is not passed to anywhere during
    // construction
  }
  // Getters, equals, hashCode, toString
}

// Marked as final - subclassing is forbidden
public final class Track {
  // Immutable object, field is final
  private final String title;

  public Track(String title) {
    this.title = title;
  }
  // Getters, equals, hashCode, toString
}
```

THREADS

The `java.lang.Thread` class is used to represent an application or JVM thread. The code is always being executed in the context of some Thread class (use `Thread#currentThread()` to obtain your own Thread).

STATE	DESCRIPTION
`NEW`	Not started.
`RUNNABLE`	Up and running
`BLOCKED`	Waiting on a monitor — it is trying to acquire the lock and enter the critical section.
`WAITING`	Waiting for another thread to perform a particular action (`notify/notifyAll`, `LockSupport#unpark`).
`TIMED_WAITING`	Same as `WAITING`, but with a timeout.
`TERMINATED`	Stopped.

Table 4: *Thread states*

THREAD METHOD	DESCRIPTION
start	Starts a Thread instance and execute its `run()` method.
join	Blocks until the `Thread` finishes.
interrupt	Interrupts the thread. If the thread is blocked in a method that responds to interrupts, an `InterruptedException` will be thrown in the other thread, otherwise the interrupt status is set.
stop, suspend, resume, destroy	These methods are all deprecated. They perform dangerous operations depending on the state of the thread in question. Instead, use `Thread#interrupt()` or a volatile flag to indicate to a thread what it should do

Table 5: *Thread coordination methods*

HOW TO HANDLE `INTERRUPTEDEXCEPTION`?

- Clean up all resources and finish the thread execution if it is possible at the current level.
- Declare that the current method throws `InterruptedException.`
- If a method is not declared to throw `InterruptedException`, the interrupted flag should be restored to true by calling `Thread.currentThread().interrupt()` and an exception, which is more appropriate at this level, should be thrown. It is highly important to set the flag back to true in order to give a chance to handle interruptions at a higher level.

UNEXPECTED EXCEPTION HANDLING

Threads can specify an `UncaughtExceptionHandler` that will receive a notification of any uncaught exception that causes a thread to abruptly terminate.

```
Thread thread = new Thread(runnable);
thread.setUncaughtExceptionHandler((failedThread,
exception) -> {
  logger.error("Caught unexpected exception in thread
      '{}'.", failedThread.getName(), exception);
});
thread.start();
```

LIVENESS

DEADLOCK

A deadlock occurs when there is more than one thread, each waiting for a resource held by another, such that a cycle of resources and acquiring threads is formed. The most obvious kind of resource is an object monitor but any resource that causes blocking (such as `wait/notify`) can qualify.

Potential deadlock example:

```
class Account {
  private long amount;

  void plus(long amount) { this.amount += amount; }

  void minus(long amount) {
    if (this.amount < amount)
      throw new IllegalArgumentException();
    else
      this.amount -= amount;
  }

  static void transferWithDeadlock(long amount,
Account first, Account second){
    synchronized (first) {
      synchronized (second) {
        first.minus(amount);
        second.plus(amount);
      }
    }
  }
}
```

The deadlock happens if at the same time:

- One thread is trying to transfer from the first account to the second, and has already acquired the lock on the first account.
- Another thread is trying to transfer from the second account to the first one, and has already acquired the lock on the second account.

Techniques for avoiding deadlock:

- Lock ordering — always acquire the locks in the same order.

```
class Account {
  private long id;
  private long amount;
  // Some methods are omitted
  static void transferWithLockOrdering(long
amount, Account first, Account second){
    boolean lockOnFirstAccountFirst = first.id <
second.id;
    Account firstLock = lockOnFirstAccountFirst  ?
first  : second;
    Account secondLock = lockOnFirstAccountFirst
? second : first;
    synchronized (firstLock) {
      synchronized (secondLock) {
        first.minus(amount);
        second.plus(amount);
      }
    }
  }
}
```

- Lock with timeout — do not block indefinitely upon acquiring the lock, but rather release all locks and try again.

```
class Account {
  private long amount;
  // Some methods are omitted

  static void transferWithTimeout(
      long amount, Account first, Account second,
      int retries, long timeoutMillis
  ) throws InterruptedException {
    for (int attempt = 0; attempt < retries;
    attempt++) {
      if (first.lock.tryLock(timeoutMillis,
      TimeUnit.MILLISECONDS)){
        try {

          if (second.lock.tryLock(timeoutMillis,
          TimeUnit.MILLISECONDS)){
            try {
              first.minus(amount);
              second.plus(amount);
            }finally {
              second.lock.unlock();
            }
          }

        }finally {
          first.lock.unlock();
        }
      }
    }
  }
}
```

The JVM can detect monitor deadlocks and will print deadlock information in thread dumps.

LIVELOCK AND THREAD STARVATION

Livelock occurs when threads spend all of their time negotiating access to a resource or detecting and avoiding deadlock such that no thread actually makes progress. Starvation occurs when threads hold a lock for long periods such that some threads "starve" without making progress.

JAVA.UTIL.CONCURRENT

THREAD POOLS

The core interface for thread pools is `ExecutorService`. `java.util.concurrent` also provides a static factory class `Executors`, which contains factory methods for the creation of a thread pool with the most common configurations.

METHOD	DESCRIPTION
`newSingleThreadExecutor`	Returns an `ExecutorService` with exactly one thread.
`newFixedThreadPool`	Returns an `ExecutorService` with a fixed number of threads.
`newCachedThreadPool`	Returns an `ExecutorService` with a varying size thread pool.
`newSingleThreadScheduled Executor`	Returns a `ScheduledExecutor Service` with a single thread.
`newScheduledThreadPool`	Returns a `ScheduledExecutor Service` with a core set of threads.
`newWorkStealingPool`	Returns an work-stealing `ExecutorService`.

Table 6: *Static factory methods*

When sizing thread pools, it is often useful to base the size on the number of logical cores in the machine running the application. In Java, you can get that value by calling `Runtime.getRuntime().availableProcessors()`.

IMPLEMENTATION	DESCRIPTION
`ThreadPoolExecutor`	Default implementation with an optionally resizing pool of threads, a single working queue and configurable policy for rejected tasks (via `RejectedExecutionHandler`), and thread creation (via `ThreadFactory`).
`ScheduledThread PoolExecutor`	An extension of `ThreadPoolExecutor` that provides the ability to schedule periodical tasks.
`ForkJoinPool`	Work stealing pool: all threads in the pool try to find and run either submitted tasks or tasks created by other active tasks.

Table 7: *Thread pool implementations*

Tasks are submitted with `ExecutorService#submit`, `ExecutorService#invokeAll`, or `ExecutorService#invokeAny`, which have multiple overloads for different types of tasks.

INTERFACE	DESCRIPTION
Runnable	Represent a task without a return value.
Callable	Represents a computation with a return value. It also declares to throw raw `Exception`, so no wrapping for a checked exception is necessary.

Table 8: *Tasks' functional interfaces*

FUTURE

Future is an abstraction for asynchronous computation. It represents the result of the computation, which might be available at some point: either a computed value or an exception. Most of the methods of the `ExecutorService` use `Future` as a return type. It exposes methods to examine the current state of the future or block until the result is available.

```
ExecutorService executorService = Executors.
newSingleThreadExecutor();
Future<String> future = executorService.submit(()
-> "result");

try {
  String result = future.get(1L, TimeUnit.SECONDS);
  System.out.println("Result is '" + result + "'.");
}
catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new RuntimeException(e);
}
catch (ExecutionException e) {
  throw new RuntimeException(e.getCause());
}
catch (TimeoutException e) {
  throw new RuntimeException(e);
}
assert future.isDone();
```

LOCKS

LOCK

The `java.util.concurrent.locks` package has a standard Lock interface. The `ReentrantLock` implementation duplicates the functionality of the synchronized keyword but also provides additional functionality such as obtaining information about the state of the lock, non-blocking `tryLock()`, and interruptible locking. Example of using an explicit `ReentrantLock` instance:

```
class Counter {
  private final Lock lock = new ReentrantLock();
  private int value;

  int increment() {
    lock.lock();
    try {
      return ++value;
    } finally {
      lock.unlock();
    }
  }
}
```

READWRITELOCK

The `java.util.concurrent.locks` package also contains a `ReadWriteLock` interface (and `ReentrantReadWriteLock` implementation) which is defined by a pair of locks for reading and writing, typically allowing multiple concurrent readers but only one writer.

```
class Statistic {
  private final ReadWriteLock lock =
new ReentrantReadWriteLock();
  private int value;

  void increment() {
    lock.writeLock().lock();
    try {
      value++;
    } finally {
      lock.writeLock().unlock();
    }
  }

  int current() {
    lock.readLock().lock();
    try {
      return value;
    } finally {
      lock.readLock().unlock();
    }
  }
}
```

COUNTDOWNLATCH

The `CountDownLatch` is initialized with a count. Threads may call `await()` to wait for the count to reach 0. Other threads (or the same thread) may call `countDown()` to reduce the count. Not reusable once the count has reached 0. Used to trigger an unknown set of threads once some number of actions has occurred.

COMPLETABLEFUTURE

`CompletableFuture` is an abstraction for async computation. Unlike plain `Future`, where the only possibility to get the result is to block, it's encouraged to register callbacks to create a pipeline of tasks to be executed when either the result or an exception is available. Either during creation (via `CompletableFuture#supplyAsync/runAsync`) or during adding callbacks (`*async` family's methods), an executor, where the computation should happen, can be specified (if it is not specified, it is the standard global `ForkJoinPool#commonPool`).

Take into consideration that if the `CompletableFuture` is already completed, the callbacks registered via non `*async` methods are going to be executed in the caller's thread.

If there are several futures you can use `CompletableFuture#allOf` to get a future, which is

completed when all futures are completed, or `CompletableFuture#anyOf`, which is completed as soon as any future is completed.

```
ExecutorService executor0 = Executors.
newWorkStealingPool();
ExecutorService executor1 = Executors.
newWorkStealingPool();

//Completed when both of the futures are completed
CompletableFuture<String> waitingForAll =
CompletableFuture
    .allOf(
        CompletableFuture.supplyAsync(() -> "first"),
        CompletableFuture.supplyAsync(() -> "second",
        executor1)
    )
    .thenApply(ignored -> " is completed.");

CompletableFuture<Void> future = CompletableFuture.
supplyAsync(() -> "Concurrency Refcard", executor0)
    //Using same executor
    .thenApply(result -> "Java " + result)

    //Using different executor
    .thenApplyAsync(result -> "Dzone " + result,
    executor1)

    //Completed when this and other future are
    //completed
    .thenCombine(waitingForAll, (first, second) -> first
    + second)

    //Implicitly using ForkJoinPool#commonPool as the
    //executor
    .thenAcceptAsync(result -> {
      System.out.println("Result is '" + result +
    "'.");
    })

    //Generic handler
    .whenComplete((ignored, exception) -> {
      if (exception != null)
        exception.printStackTrace();
    });

//First blocking call - blocks until it is not finished.
future.join();

future
    //Executes in the current thread (which is main).
    .thenRun(() -> System.out.println("Current thread
    is '" + Thread.currentThread().getName() + "'."))

    //Implicitly using ForkJoinPool#commonPool as the
    //executor
    .thenRunAsync(() -> System.out.println("Current" +
    "thread is '" + Thread.currentThread().getName() +
    "'."));
```

CONCURRENT COLLECTIONS

The easiest way to make a collection thread-safe is to use `Collections#synchronized` family methods. Because this solution performs poorly under high contention, `java.util.concurrent` provides a variety of data structures which are optimized for concurrent usage.

LIST

IMPLEMENTATION	DESCRIPTION
`CopyOnWriteArrayList`	It provides copy-on-write semantics where each modification of the data structure results in a new internal copy of the data (writes are thus very expensive, whereas reads are cheap). Iterators on the data structure always see a snapshot of the data from when the iterator was created.

Table 9: *Lists in java.util.concurrent*

MAPS

IMPLEMENTATION	DESCRIPTION
`ConcurrentHashMap`	It usually acts as a bucketed hash table. Read operations, generally, do not block and reflect the results of the most recently completed write. The write of the first node in an empty bin is performed by just CASing (compare-and-set) it to the bin, whereas other writes require locks (the first node of a bucket is used as a lock).
`ConcurrentSkipListMap`	It provides concurrent access along with sorted map functionality similar to `TreeMap`. Performance bounds are similar to `TreeMap` although multiple threads can generally read and write from the map without contention as long as they are not modifying the same portion of the map.

Table 10: *Maps in java.util.concurrent*

SETS

IMPLEMENTATION	DESCRIPTION
`CopyOnWriteArraySet`	Similar to `CopyOnWriteArrayList`, it uses copy-on-write semantics to implement the Set interface.
`ConcurrentSkipListSet`	Similar to `ConcurrentSkipListMap`, but implements the Set interface.

Table 11: *Sets in java.util.concurrent*

Another approach to create a concurrent set is to wrap a concurrent map:

```
Set<T> concurrentSet = Collections.newSetFromMap(
new ConcurrentHashMap<T, Boolean>());
```

QUEUES

Queues act as pipes between "producers" and "consumers." Items are put in one end of the pipe and emerge from the

other end of the pipe in the same "first-in first-out" (FIFO) order. The `BlockingQueue` interface extends `Queue` to provide additional choices of how to handle the scenario where a queue may be full (when a producer adds an item) or empty (when a consumer reads or removes an item). In these cases, `BlockingQueue` provides methods that either block forever or block for a specified time period, waiting for the condition to change due to the actions of another thread.

IMPLEMENTATION	DESCRIPTION
`ConcurrentLinkedQueue`	An unbounded non-blocking queue backed by a linked list.
`LinkedBlockingQueue`	An optionally bounded blocking queue backed by a linked list.
`PriorityBlockingQueue`	An unbounded blocking queue backed by a min heap. Items are removed from the queue in an order based on the Comparator associated with the queue (instead of FIFO order).
`DelayQueue`	An unbounded blocking queue of elements, each with a delay value. Elements can only be removed when their delay has passed and are removed in the order of the oldest expired item.
`SynchronousQueue`	A 0-length queue where the producer and consumer block until the other arrives. When both threads arrive, the value is transferred directly from producer to consumer. Useful when transferring data between threads.

Table 12: *Queues in java.util.concurrent*