1. Методы: toString: Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса. hashCode: Метод hashCode позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Получение типа объекта getClass: Метод getClass позволяет получить тип объекта. Equals: Метод equals сравнивает 2 объекта на равенство. Метод equals принимает в качестве параметр объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Реализация данной функции по умолчанию просто проверяет по ссылкам два объекта на предмет их эквивалентности.
2. функция hashCode () для объекта возвращает номер ячейки памяти, где объект сохраняется (данное значение называется хэш-значение). Поэтому, если изменение в код приложения не вносятся, то функция должна выдавать одно и то же значение. При незначительном изменении кода значение hashCode также изменится. Так же хэш код бита обычно совпадает с его юникодом. Значение hashCode зависит от реализации. например, String класс реализует функцию hashCode() в зависимости от значения. это означает что String a=new String("b"); String b=new String("b"); будет иметь тот же hashCode, но это два разных объект, и a==b вернется false.

С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, у нас уже есть готовый класс Person он имеет поля возраст, голод и другие теперь нам нужно создать класс Student куда практичней создать его на основе Person потому что он будет иметь те же поля что и Person + свои. Private поля и методы не наследуется они доступны только в том классе где их создали.

1. this представляет текущий экземпляр класса, в то время как super - текущий экземпляр родительского класса. super()используется для вызова конструктора без аргументов, или как его ещё называют, конструктора по умолчанию родительского класса. this и super в Java используются для обращения к переменным экземпляра класса и его родителя. Вообще-то, к ним можно обращаться и без префиксов super и this, но только если в текущем блоке такие переменные не перекрываются другими переменными, т.е. если в нем нет локальных переменных с такими же именами, в противном же случае использовать имена с префиксами придется обязательно. this() и super non-static переменные так что их нельзя использовать в main и других static методах. Внутри конструктора this и super должны стоять выше всех других выражений, в самом начале, иначе компилятор выдаст сообщение об ошибке. Из чего следует, что в одном конструкторе не может быть одновременно и this(), и super().
2. Ссылка базового класса может ссылаться на объекты своих производных типов, но наоборот не работает потому что базовый класс может не обладать методами производного типа. Так же объект производного класса может быть использован вместо объекта базового, но не наоборот опять из-за того, что базовый может не иметь того же метода что и производный.
3. Это когда мы берем метод класса, от которого мы его унаследовали и меняем то что он делает. Нам это нужно, например, для того чтобы не создавать новый метод с нуля, а внести легкое изменение в уже готовый, например, у нас есть метод Voice для собаки и кота он будет разный, но только в выводимом значение поэтому мы можем унаследовать его от класса Animal и переопределить для кота и собаки. При переопределении мы можем изменить:
   1. Модификатор доступа, если расширять (package -> protected -> public)
   2. Возвращаемый тип если выполняется Downcasting (нисходящее преобразование, преобразование вниз по иерархии)
   3. 4) Имя аргументов

Возможно вовсе убрать секцию throws в методе, так как она уже определена.

Так же, возможно добавлять новые исключения, которые наследуются от объявленных или исключения времени выполнения.

Но мы не можем изменить количество аргументов или их порядок так как в таком случае будет перегрузка

Переопределять можно только методы классов родителей или суперклассов.

1. А Статические методы не могут быть переопределены в точном смысле слова, но они могут скрыть родительские статические методы. На практике это означает, что компилятор будет решать, какой метод выполнять во время компиляции, а не во время выполнения, как это происходит с переопределенными методами экземпляра. Переопределение:

Переопределение в Java просто означает, что конкретный метод будет вызываться на основе типа времени выполнения объекта, а не типа его времени компиляции (что имеет место с переопределенными статическими методами).

Скрытие: Статические методы родительского класса не являются частью дочернего класса (хотя они доступны), поэтому не стоит переопределять его. Даже если вы добавите другой статический метод в подкласс, идентичный методу в его родительском классе, этот статический метод подкласса является уникальным и отличается от статического метода в его родительском классе

В случае методов экземпляра вызывается метод фактического класса объекта.

1. Обозначая метод класса модификатором final, мы имеем ввиду, что ни один производный класс не в состоянии переопределить этот метод, изменив его внутреннюю реализацию. Класс, помеченный как final, не поддается наследованию и все его методы косвенным образом приобретают свойство final. Применение признак, а final в объявлениях классов и методов способно повысить уровень безопасности кода. Если класс снабжен модификатором final, никто не в Состоянии расширить класс и, вероятно, нарушить при этом его контракт (контракт — это когда мы точно знаем, что, например, equals должен совершить сравнение по всем полям и ничто не в силах этому помешать) Использование модификатора final в объявлениях классов способствует также повышению эффективности некоторых операций проверки типов. В этом случае многие подобные операции могут быть выполнены уже на стадии компиляции и поэтому потенциальные ошибки выявляются гораздо раньше. Если компилятор встречает в исходном тексте ссылку на класс final, он может быть "уверен", что соответствующий объект относится именно к тому типу, который указан.
2. Существует расширяющее и сужающее приведение.

Когда мы напишем:

Animal animalCat = new Cat();

Animal animalDog = new YorkshireTerrier();

Это расширяющее приведение (или неявное). Мы расширили ссылки animalCat и animalDog. Они ссылаются на объекты Cat и Dog. При таком приведении мы не можем через ссылку animalCat/animalDog вызвать методы, которые есть в Cat/Dog, но которых нету в Animal.

Сужающее приведение (или явное) происходит в обратную сторону:

Animal animalCat = new Cat();

Animal animalDog = new YorkshireTerrier();

Cat cat = (Cat) animalCat;

YorkshireTerrier dog = (YorkshireTerrier) animalDog;

В данном случае мы явно указали к какому типу хотим привести данный объект.

Опасность явного приведения в том что мы можем написать

YorkshireTerrier dog = (YorkshireTerrier) animalCat; и компилятор примет это а вот во время исполнения произойдет ошибка. Она произойдет из-за того, что RunTime видит, что Cat и YorkshireTerrier два разных класса избежать этого можно используя команду instanceof если не знаете, что будет дальше или просто не делать явное приведение. Приведение нам нужно тогда, когда мы хотим собрать несколько классов наследующихся от одного и вызвать общий метод для них.

1. getClass() возвращает объект Сlass который содержит некоторые методанные о классе:
   1. название
   2. пакет
   3. методы
   4. поля
   5. конструкторы
   6. аннотации

так же можно провести некоторые полезные проверки, например, можно написать так obj.getClass(). isAnnotationPresent(Processable.class) данная строка кода смотрит на, то есть ли у объекта класса аннотация Processable. каждый объект в java имеет (принадлежит) класс и имеет соответствующий объект Class, который содержит метаданные о нем, который доступен во время выполнения. Объект Class является своего рода мета-объектом, описывающим класс объекта (чертеж реального класса).

Оператор instanceof нужен, чтобы проверить, был ли объект, на который ссылается переменная X, создан на основе какого-либо класса Y. Он нам может пригодится для проверки можно ли привести один ссылочный тип к другому.

Instanceof возвращает значение типа boole а getClass возвращает объект Class.

Как мы знаем конструктор базового класса, если он есть, всегда вызывается первым при создании любого объекта. Instanceof руководствуется именно этим принципом, когда пытается определить, был ли объект А создан на основе класса Б. Если конструктор базового класса вызван, значит никаких сомнений быть не может.

Ссылка про inscenceof <https://javarush.ru/groups/posts/2018-kak-rabotaet-operator-instanceof>

instanceof проверяет, является ли ссылка объекта в левой части (LHS) экземпляром типа в правой части (RHS) или некоторым подтипом.

getClass() == проверяет, идентичны ли типы.

1. Правила переопределения данных методов весьма просты и существуют потому что это контрактные методы класса Object. Equals:
   1. Рефлексивность-для любого заданного значения x, выражение x.equals(x) должно возвращать true. x!=0
   2. Симметричность-для любых заданных значений x и y, x.equals(y) должно возвращать true только в том случае, когда y.equals(x) возвращает true.
   3. Транзитивность-для любых заданных значений x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, x.equals(z) должно вернуть значение true.
   4. Согласованность-для любых заданных значений x и y повторный вызов x.equals(y) будет возвращать значение предыдущего вызова этого метода при условии, что поля, используемые для сравнения этих двух объектов, не изменялись между вызовами.
   5. Сравнение null-для любого заданного значения x вызов x.equals(null) должен возвращать false.

Только в случае если ваш переопределенный метод соответствует всему перечисленному считается что вы выполнили контракт.

HashCode:

1. Вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать одно и то же хэш-значение, при условии, что поля объекта, участвующие в вычислении значения, не изменялись.
2. вызов метода hashCode над двумя объектами должен всегда возвращать одно и то же число, если эти объекты равны (вызов метода equals для этих объектов возвращает true).
3. Вызов метода hashCode над двумя неравными между собой объектами должен возвращать разные хэш-значения. Хотя это требование и не является обязательным, следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.

Методы HashCode и equals необходимо переопределить попарно потому что если один переопределить, а второй нет то может случится так что по методу equals они будут равны, а вот не переопределённый HashCode будет считать, что это 2 разных объекта и поместив некий объект в hash таблицу мы рискуем его не получить обратно. В случае если мы определим метод Hashcode, а equals нет то мы все еще не сможем найти наш объект в hash таблице так как для успешного поиска объекта в хэш-таблице помимо сравнения хэш-значений ключа используется также определение логического равенства ключа с искомым объектом. Т. е. без переопределения метода equals никак не получится обойтись.

toString:

1. Должен вызываться для всех полей объекта.
2. Всегда возвращает String
3. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. При объявлении абстрактных классов используется ключевое слово abstract и использовать конструкторы для создания объекта абстрактного класса нельзя, произойдёт ошибка если попытаться потому что это нарушит саму мысль абстрактного класса так же в абстрактном классе иногда отутвует реализация некоторых методов. В следствие чего можно понять, что у методов в абстрактном классе тело может быть, а может и не быть. Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный. Абстрактный класс может содержать внутри себя не абстрактные методы, но абстрактные методы могут быть только в абстрактном классе. Абстрактные классы нужны нам, когда у нас есть несколько классов для которых было бы удобно сделать общие поля и методы, которые будут немного отличатся в таком случаем мы используем абстрактный класс и переопределяем его методы. Да, в абстрактном классе в Java можно объявить и определить конструкторы. Поскольку создавать экземпляры абстрактных классов нельзя, вызвать такой конструктор можно только при формировании цепочки конструкторов, то есть при создании экземпляра конкретного класса-реализации.
4. Интерфейс описывает поведение, которым должны обладать классы, реализующие этот интерфейс. «Поведение» — это совокупность методов. Cоздание интерфейса очень похоже на создание обычного класса, только вместо слова class мы указываем слово interface. Определить интерфейс можно по ключевому слову interface и по отсутвию тела у методов данного класса. Для того чтобы воспользоваться методом определенном в интерфейсе нужно имплементировать интерфейс и реализовать его метод. Классы имплементирующие методы класса интерфейса должны реализовывать методы интерфейса иначе будет ошибка. Так же преимущество интерфейсов в том, что множественного наследования в java нет, а вот интерфейсов класс может реализовывать сколько захочет. Методы в классе интерфейсе могут иметь спецификатор default что будет указывать на то что данный метод не нужно реализовывать, и он будет общих у всех. В интерфейсе нельзя реализовывать конструкторы Анонимные классы могут реализовывать интерфейсы без ключевого слова implements. Когда вы объявляете интерфейс, вы объявляете новый ссылочный тип данных. Вы можете использовать название интерфейса в качестве типа данных так же, как и любые другие типы. Если вы объявляете переменную типа интерфеса, то вы можете можете присвоить ей объект любого класса, который реализует этот интерфейс. Переопределять интерфейсы нельзя (вдруг резко решили добавить новый метод) нельзя потому что классы, которые уже реализовали интерфейс сломаются так что лучше определить новый интерфейс, который расширит старый.
5. Clone делает по битовую копию. Интерфейс Clonable нужен для создания клонов которые смогли бы существовать отдельно от оригинала. Например, при обыкновенном присваивание объектов (obj1 = obj2;) передаются ссылки на объект. В итоге два экземпляра ссылаются на один объект, и изменение одного приведет к изменению другого, а после использовании команды clone у нас будет 2 независимых версии объекта. метод clone () может выбрасывать исключение CloneNotSupportedException. Данное исключение возникает в случае, когда клонируемый класс не имеет реализации интерфейса Cloneable. Интерфейс Cloneable не реализует ни одного метода. Он является всего лишь маркером, говорящим, что данный класс реализует клонирование объекта. Само клонирование осуществляется вызовом родительского метода clone(). Например, class User implements Cloneable после этого мы можем реализовать метод clone **public** User clone() **throws** CloneNotSupportedException {

User clone = (User)**super**.clone();

**return** clone;

      }

При помощи которого копировать экземпляры класса User и делать с клонами что хотим без изменения оригинала. Для того чтобы клонировать поля класса, к которому мы обращаемся мы должны вызывать команду клон на класс через команду super. В случае если мы хотим добавить некое значение нашему клону из библиотеки и прировнять ее нашему полю мы должны написать так clone.birthday = (GregorianCalendar) birthday.clone(); где GregorianCalendar библиотека, а birthday поле нашего клона

1. Если мы хотим сравнить и отсортировать объекты некого класса они должны применять интерфейс Comparable<E>. При применении интерфейса он типизируется текущим классом. Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

Пример compareTo: работает только по 1 полю.

public int compareTo(Person p){

    return name.length()-p.getName().length();

}

Так же существует более гибкий способ на случай если разработчик не реализовал Comparable это интерфейс Comparator<E>. интерфейс Comparator содержит ряд методов, ключевым из которых является метод compare():

Пример compare: сколько угодно полей сортировать

public int compare(Person a, Person b){

 return a.getName().compareTo(b.getName());

   }

Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс.

Используйте Comparable:

1. если объект находится под вашим контролем.
2. если поведение сравнения является основным поведением сравнения.

Используйте Comparator:

1. если объект находится вне вашего контроля, и вы не можете заставить их реализовать Comparable.
2. если вы хотите сравнить поведение, отличное от по умолчанию (которое указано в Comparable).