Object Oriented Programming

(type parameter, generics, ArrayList)

Type safe method call

Method parameters – type safe, checks made by compiler and JVM

- Return type specifies type of return value
- Type check of return value is made in compile time
- Type of formal parameters specify actual parameter types
- Matching formal and actual parameters are made in compile and run-time
    - Number
    - Type

Method for generic purpose

What to do when same operation has to be executed on different types?

Generalization

- Same behavior with type dependent operation → overload
- Behavior with type independent operation
  - Base subject class – Object
  - Generic – type parameter

Type dependent operations

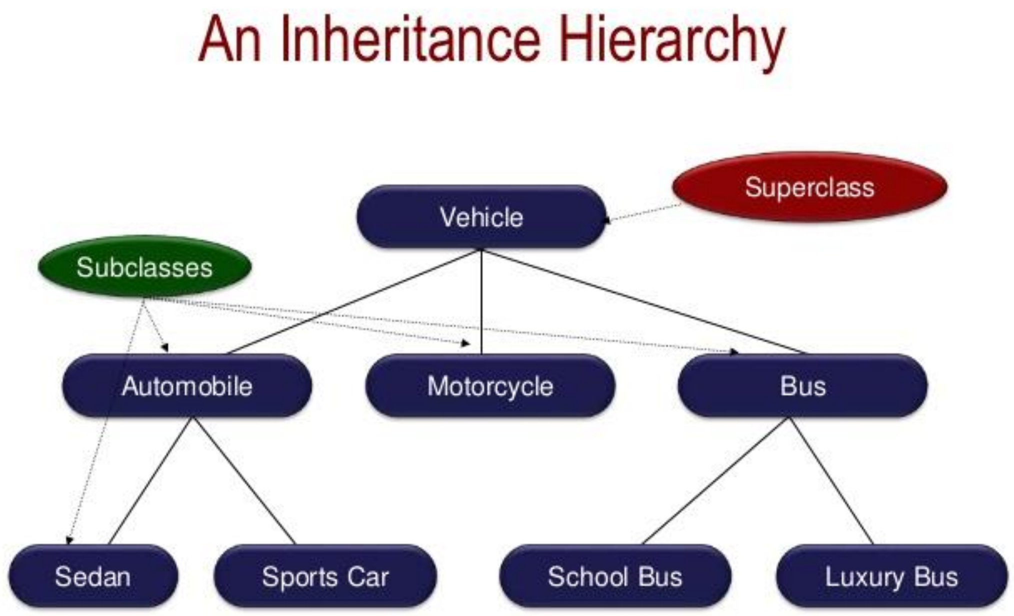
- Handler method **does** access components of subject class
- Type safe access is required to provide appropriate interface
- Parameter type specified at compile time
- Functionality:
  - Parameters of different types are used for the same logical operation – behavior
- Method: **Method overload**

Type independent operations

- Basically handler class does **not** access components of subject class
- For example: enhanced *general storage* only
- Functionality:
  - Reusable for more classes
  - Extensible and reducible storage size (unlike arrays)
  - Built in sort method
  - *Using hash of contents just to accelerate search*

Type independent – base class

- A base class can be used as subject
- All subclasses can be casted to this
- Interface of common type is accessible
- Root of class hierarchy is `Object`



Upcast

- Casting up on class hierarchy tree until the root – valid
- Any object can be casted to Object implicitly or explicitly
- This cast changes the type of reference
- This cast **does not** change type of referenced object

Upcast example

```
Car car = new Car();
Vehicle vehicle = car;                              // Valid
Object vehicleObject = car;         // Valid

// Reserved object is still a Car,
// but can be accessed via Car, Vehicle and Object interfaces,
// specified by references above
```

Base class storage

```
Vehicle[] cars = new Vehicle[10];
cars[0] = new Car();
cars[2] = new Car();
```

```
Vehicle[] buses = new Vehicle[10];
buses[0] = new Bus();
buses[2] = new Bus();
```

```
Object[] trucks = new Object[10];
trucks[0] = new Truck();
trucks[1] = new Truck();
```

Base class storage – combined

```
Vehicle[] vehicles = new Vehicle[10];
vehicles[0] = new Car();
vehicles[1] = new Bus();
vehicles[2] = new Truck();
```

```
Object[] objects = new Object[10];
objects[0] = vehicles[0];
objects[1] = vehicles[1];
objects[2] = vehicles[2];
```

Downcast

- Implicit cast down or side from reference in class hierarchy tree is NOT valid
- Object reference can be casted **explicitly** to any class – compile time
- Explicit cast is type checked at run time → `ClassCastException`
- **Cast is valid**, when interface of object **contains** required interface:
  - Downcast on the **same branch**
  - Downcast **until** the type of the allocated **object** type

Downcast example

```
Vehicle[] vehicles = new Vehicle[10];
vehicles[0] = new Car();
Object[] objects = new Object[10];
objects[0] = vehicles[0];
```

```
Car car = (Car)vehicles[0];
Car car = (Car)objects[0];
SportCar car = (SportCar)vehicles[0];     //Invalid
SportCar car = (SportCar)objects[0];      //Invalid
```

How to use stored objects?

```
if(objects[0] instanceof Car) {
        Car car = (Car)objects[0];
}
if(objects[0] instanceof Bus) {
        Bus bus = (Bus)objects[0];
}
…
```

Check the leaf classes!

```
<type>.class.isAssignableFrom(<object>)
```

Really generic implementation

To create a really generic implementation programmer has to

- Create many implementations for different base classes

Or

- Use the root class (Object) as storage type

Generic implementations

Implementations by type

- **Type safe in compile time, parameter types can be checked**
- **Interface of stored objects is specified**
- Multiple implementations

Base subject class

- Compile time type safety is lost, everything is an Object
- Interface of stored objects is not specified
- **Single implementation**

Type safety with single implementation

- Combine benefits of pervious appearances
  - Type safe at compile time
  - Specification of subject interface at compile time
  - Single implementation to handle many types
- Type **parameter is introduced**
  - Declared and checked in compile time

Type parameter

- A placeholder in declaration
- Can be used in method declaration and body
- Set at the call of a generic method but in **compile time**
- Evaluated at compile time
- Type safety is guaranteed at run time
- Can **not** be primitive type
- Can be **inferenced** from method call

The code of a method is generic for all types, but the method does not work with any times. The subject type has to specified at compile time.

Type parameter

Methods can have type parameters to make them generic

Type parameter section

- Precedes the return type
- Delimited by angle brackets ”<” and ”>”
- Contain one or more type parameter, separated by comas ”,”
- Type parameters can be used anywhere in method declaration

Method declaration

```
<visiblity> [type parameters] <return type> <method name>([formal
parameters])
{
   <method body>
}
```

```
public <T> void print(T item) {
   System.out.println(item);
}
```

Generic method – example

```
public <T> T getItem(T item) {
   return item;
}

Car car = getItem(new Car());        //No effect, returns the object
// Type parameter is inferred from method argument
// Type of variable and return type is checked at compile time
```

Type parameter constraints

- When handler class wants to **use** services/components of instance of subject class, accessing interface has to be known
- **Restrictions** have to be defined about subject class
- Specification of **interface** (class) to use in class hierarchy

Restricted Generic method declaration

```
<visiblity>
    [<type parameter> extends <superclass>]
    <return type> <method name>([formal parameters])
{
    <method body>
}
```

Generic method – bounded type

```
public <T extends ClassWithValue> T incrementValue(T item) {
      item.setValue(item.getValue() + 1);
   return item;
}

Product product = incrementValue(new Product());
//Product is a subclass of ClassWithValue
```

Generic class

Collection of generic methods can be encapsulated in a generic class

Generic class

- All methods work with same type(s) given as parameter(s)
- Type parameters set for the whole class at compile time

Generic class example

```
class Storage<T> {
	public final int STORAGE_SIZE = 10;

	private T[] storage = new T[STORAGE_SIZE];
	private int lastIndex = 0;

	public void addItem(T newItem) { storage[lastIndex] = 
newItem; }
	public T getItem(int index) { return storage[index]; }
}
```

Generic class usage example

```
Storage<Car> carStorage = new Storage<Car>();
carStorage.addItem(new Car());
carStorage.addItem(new Car());

Storage<Bus> busStorage = new Storage<Bus>();
busStorage.addItem(new Bus());
busStorage.addItem(new Bus());
```

Generic class usage example – up-/downcast

```
Storage<Vehicle> vehicleStorage = new Storage<Vehicle>();

vehicleStorage.addItem(new Car());
vehicleStorage.addItem(new Bus());

Bus bus = (Bus)vehicleStorage.getItem(1);
Bus otherBus = (Bus)vehicleStorage.getItem(0); //Invalid!!!
```

Generic List class

```
public class ArrayList<T> { … }
```

Developer documentations

(https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Genereic list methods

- clear – clear the list
- add – add new item
- contains – check if list contains item (by reference, not by value)
- get – get item
- isEmpty – check if list is empty
- remove – remove item
- set – set item
- size – get list size

Usage of generic lists

```
ArrayList<Vehicle> vehicles = new ArrayList<Vehicle>();

vehicles.add(new Car());
vehicles.add(new Bus());
vehicles.add(new Truck());

// Downcasts have to be checked by programmer
```

Agile development

Functional requirements

User story:

- Initiated in problem domain, by problem experts (customer)
- Natural language description of one feature
- Informal description of a feature
- Written from a perspective of a user

User story templates

- As a <role> I can <capability>, so that <receive benefit>
- As <who> <when> <where>, I <want> because <why>
- As <persona>, I can <what?> so that <why?>

User story example

- As an *administrator* I want to add car/bus/truck to the site to maintain a collection
- As a *salesman* I want to select an available car/bus/truck to lend
- As a *salesman* I want to lend a car/bus/truck to see which is available
- As a *salesman* I want to retrieve a a car/bus/truck to get its rental price
- As a *transport manager* I want to send car/bus/truck to inner transport
- As an *owner* I want to see the distance vehicles performed to order service
- As an *owner* I want to see states of vehicles to report utilization

System requirements

- A **formal** description of system requirements
- Derived from user stories
- Additial IT requirements – security, integration, storage,
- By IT experts
- Accepted by both customer and developer sides

Technical specification

- A framework of implementation
- Created by development leaders
- In our case should be a result of a developer workshop

Tasks / issues – back log

- Development task created from
  - User stories
  - System specifications
  - Technical specifications
- Not equal to user stories
- Also contain framework implementations and changes

Tasks - example

1. Create Car/Bus/Truck classes
2. Refactor classes to reuse generic components
3. Create vehicle storage
4. Create Company class with storage and vehicle administration
5. Set general parameters from console input
6. Refactor Storage class to use ArrayList instead of array
7. Extend Company class with rental functions (find, lend, retrieve)

Scrum poker

- Select a **base unit** for complexity
- Change of caption/color/text has **0** (zero) complexity
- Complexity: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
- Select a **threshold** for splitting (creating subtasks)

Subtasks

When estimated complexity is above the selected threshold, task has to be split into subtasks, complexity of subtasks have to be estimated.

Task #7 has to be split into subtasks:

- Find available vehicle of type
- Lend / send to transport
- Retrieve with cost calculation

Sprint kick-off

- Select tasks to solve in the sprint – **sort backlog** by stakeholder by importance and complexity
- **Select first N** items based on planned developer performance
- First underestimate performance of developers
- Extend sprint scope if run out of tasks
- At end of sprint, the outcome is a deliverable product

- Practice scope: solve all remainig tasts

Sprint scope and goals

- Create 1-3 sprint goals based on selected tasks
- Sprint goals are NOT subject of change through the sprint
- Deliver an application with basic administration functionality, but no statistics

Implementations

- Implement all tasks in different branches of the repository
- Merge to the appropriate main branch when a task has been finished
- Create developer documentation only when implementations is NOT straightforward
- Do not store developer documentation in code
- Link questionable implementation part amd developer documentation to specifications

Daily stand-up

Each participants

- Describe progress since previous stand-up
- Compare progress to the plans
- Describe planned progress until next stand-up
- List required resources with their availability
- Report estimation errors, modify sprint scope if necessary
- Ask for implementation help, but explanation is subject of another meeting

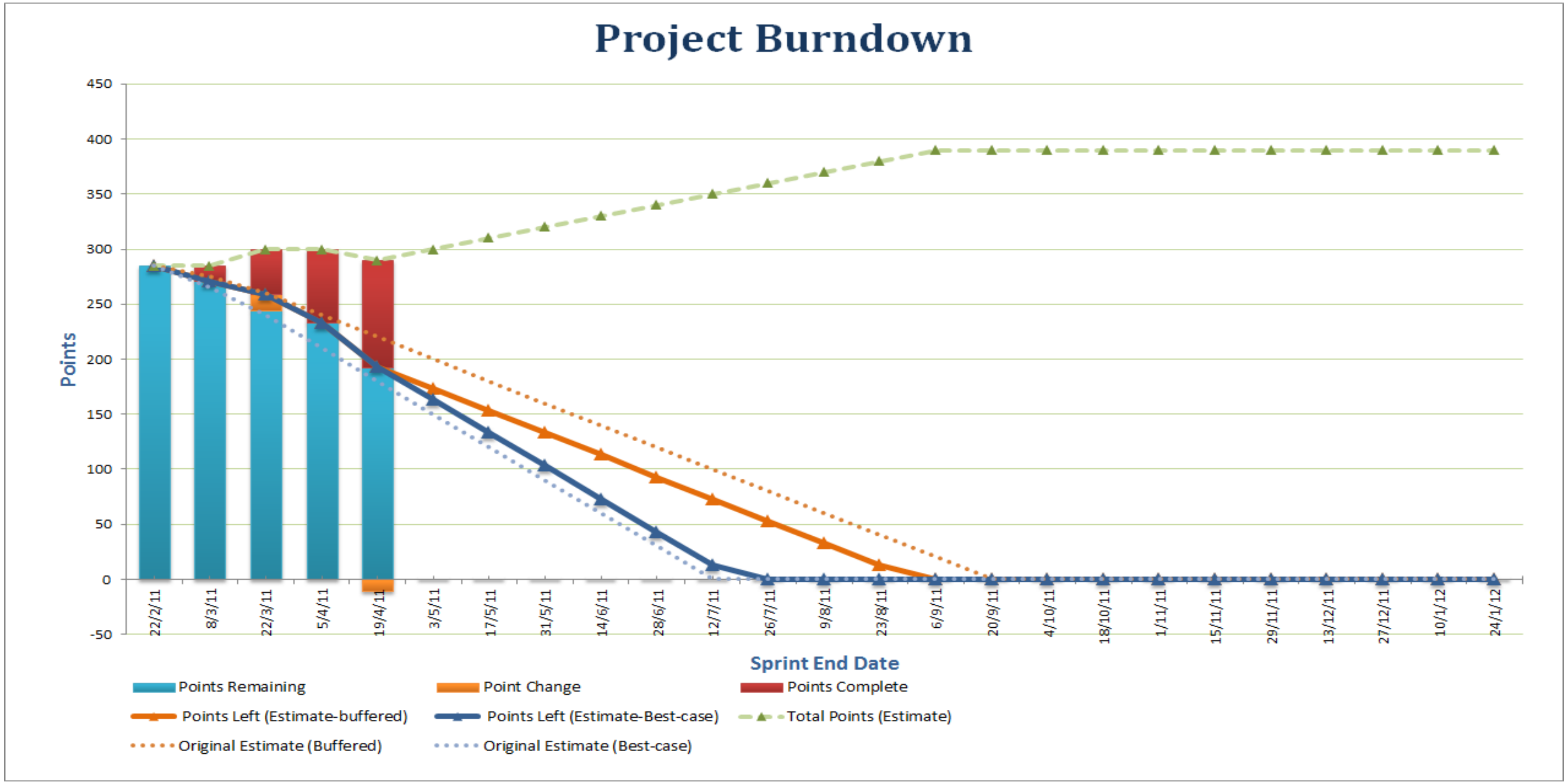
Sprint scope change

- If performance underestimated, sprint scope can be extended
- If task can not be solved because of unavailble resources, task can be replaced by another one, with the same complexity
- Sprint scope can not be shrinked

Burn down chart

- Shows progress throug sprint
- Manager/stakeholder check fulfillment of planned progress day-by-day
- When remaining tasks are above plans, more resources are needed

Burndown chart example



Sprint finish

At end of sprint

- Performance can be measured for all developers
- Unfinished tasks moved to back log
- Burndown published
- Presentation of product to see fulfillment of requirements

Sprint retrospection

- Find what was GOOD in sprint
- Find what has to be enhanced
  - Check fulfillment of sprint goals
  - Check estimation accuracy – find reasons of under/over estimations
  - Find reasons of unfinished tasks
  - Check developer performance gradients – investigate reasons
- Create actions to increase development quality

When project finished

- Be HAPPY
- Be SATISFIED
- Be PROUD

Meanwhile

- Prepare for maintenance
- Enhance sustainability